VŠB – Technická univerzita Ostrava
Fakulta elektrotechniky a informatiky
Katedra informatiky

Měření pokrytí kódu pomocí metriky „pokrytí cest“

Code Coverage Meassurement based on Path Coverage Metric

2018 Bc. Daniel Mrózek

VŠB - Technická univerzita Ostrava
Fakulta elektrotechniky a informatiky
Katedra informatiky

Zadání diplomové práce

Student: **Bc. Daniel Mrózek**

Studijní program: N2647 Informační a komunikační technologie

Studijní obor: 2612T025 Informatika a výpočetní technika

Téma: Měření pokrytí kódu pomocí metriky „pokrytí cest“
Code Coverage Meassurement based on Path Coverage Metric

Jazyk vypracování: čeština

Zásady pro vypracování:

Cílem práce je vytvořit podporu do prostředí Eclipse, která umožní měření pokrytí zdrojového kódu pomocí metriky „pokrytí cest“ podobným způsobem, jako se měří jiné metriky (například metrika pokrytí větví) v nástroji EclEmma.

Postup pro vypracování:
1. Seznamte se s problematikou metrik pro měření pokrytí kódu testy.
2. Seznamte se s nástrojem EclEmma a prozkoumejte možnosti rozšíření tohoto nástroje.
3. Seznamte se se současným stavem, který byl dosažen v rámci diplomové práce [1].
4. Navrhněte a dle návrhu zajistěte možnost vytvářet grafy cest pro zdrojový kód.
5. Navrhněte a vytvořte podporu pro měření pokrytí kódu dle dané metriky. Při realizaci se soustřeďte na vhodné znázornění nepokrytých cest vůči zdrojovému kódu.

Práce bude zejména obsahovat:
1. Stručné shrnutí metrik pokrytí kódu.
2. Návrh rozšíření nástroje EclEmma nebo nástroje hotového v rámci diplomové práce [1].
3. Naimplemtovaný návrh rozšíření, který bude dostupný v repozitáři aplikaci git.cs.vsb.cz a bude sestavitelný pomocí systému maven či gradle.
4. Sestavené rozšíření, které bude možno instalovat skrze mechanismus "Eclipse updatesite" nebo "Eclipse marketplace"

Seznam doporučené odborné literatury:

[1] MICHÁLEK, Jan. Systém pro měření pokrytí kód testy [online]. 2017 [cit. 2017-10-10]. Dostupné z: http://hdl.handle.net/10084/119048. Diplomová práce. Vysoká škola báňská - Technická univerzita Ostrava.BLEWITT, Dr Alex, 2014. Mastering Eclipse Plug-in Development. B.m.: Packt Publishing - ebooks Account. ISBN 978-1-78328-779-6.
[2] VOGEL, Lars a Mike MILINKOVICH, 2015. Eclipse Rich Client Platform. 3 edition. B.m.: Lars Vogel. ISBN 978-3-943747-13-3.

Dále dle doporučení vedoucího diplomové práce.

Formální náležitosti a rozsah diplomové práce stanoví pokyny pro vypracování zveřejněné na webových stránkách fakulty.

Vedoucí diplomové práce: **Ing. Jan Kožusznik, Ph.D.**

Datum zadání: 01.09.2017

Datum odevzdání: 30.04.2018

doc. Ing. Jan Platoš, Ph.D.	prof. Ing. Pavel Brandštetter, CSc.
vedoucí katedry	*děkan fakulty*

Prohlašuji, že jsem tuto diplomovou práci vypracoval samostatně. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

V Ostravě 25. dubna 2018 .....

Rád bych na tomto místě poděkoval všem, kteří mi s prací pomohli, protože bez nich by tato práce nevznikla. Zvláště mému vedoucímu Ing. Janu Kožusznikovi, Ph.D., který mě stále motivoval a celou dobu správně směřoval. Vždycky si taky našel čas ke konzultaci.

Abstrakt

Cílem této diplomové práce je vytvořit podporu do prostředí Eclipse, která umožní měření pokrytí zdrojového kódu pomocí metriky pokrytí cest. V řešení byl použit prototyp nástroje, který pracuje právě s touto metrikou. K prezentaci výsledků pak novější technologie Eclipse 4. Sestavení a distribuce výsledků, byla provedena pomocí nástrojů Maven a Tycho. Vytvořené řešení poskytuje vykreslení cest v prostředí Eclipse, využitím zmíněné metriky a lze jej zde instalovat pomocí standardního Eclipse P2 mechanizmu. Hlavním výsledkem je tedy nástroj do prostředí Eclipse, využívající k měření kódu metriku pokrytí cest.

Klíčová slova: Eclipse, E4, pokrytí cest, JUnit, Java, JaCoCo, Maven, Tycho, P2

Abstract

The aim of this diploma thesis is to create support for the Eclipse environment, which will allow the measurement of the source code coverage using path coverage metric. The solution use a prototype tool that work with this metric. To present the results, the newer Eclipse 4 technology is used. The assembly and distribution of the results was done using the Maven and Tycho tools. The solution created provides Eclipse path rendering by using the metric and can be installed in the environment using the standard Eclipse P2 mechanism. The main result is the Eclipse tool, which uses the route coverage metric to measure the code.

Key Words: Eclipse, E4, path coverage, JUnit, Java, JaCoCo, Maven, Tycho, P2

Obsah

Seznam použitých zkratek a symbolů

API	–	Application Programming Interface
AST	–	Abstract Syntax Tree
BC	–	Branch coverage
CFG	–	Control flow graph
CSV	–	Comma-separated values
DOM	–	Document Object Model
EMF	–	Eclipse Modeling Framework
HTML	–	HyperText Markup Language
HTTP	–	Hypertext Transfer Protocol
IDE	–	Integrated Development Environment
IP	–	Internet Protocol
JDK	–	Java Development Kit
JDT	–	Java Development Tools
JVM	–	Java Virtual Machine
MVC	–	Model-view-controller
OSGi	–	Open Services Gateway initiative
PC	–	Path coverage
PDE	–	Plug-in Development Environment
POJO	–	Plain Old Java Object
POM	–	Project Object Model
REST	–	Representational State Transfer
SC	–	Statement coverage
SW	–	Software
SWT	–	Standard Widget Toolkit
TCP	–	Transmission Control Protocol
UI	–	User Interface
URI	–	Uniform Resource Identifier
URL	–	Uniform Resource Locator
XML	–	Extensible Markup Language

Seznam obrázků

Seznam výpisů zdrojového kódu

1 Úvod

V testování softwaru se pokrytím kódu měří počet řádků zdrojového kódu provedeného během dané testovací sady. Výsledek pak vyjadřuje procentuální pokrytí kódu v závislosti na použité metrice. Tyto informace pak můžou být využity ke zlepšení jednotlivých testů či odhalení možných chyb v testovaném kódu. Nástroje umožňující měření pokrytí kódu pak aplikují konkrétní metriky.

Po prozkoumání aktuální nabídky nástrojů pro prostředí Eclipse(„Eclipse marketplace“ [2] a „Eclipse Updatesite“[6]) zabývajících se oblastí měření pokrytí kódu bylo zjištěno, že většina nalezených nástrojů využívá metriku pokrytí skoků/rozhodování. Žádný nalezený nástroj pak nepodporoval metriku pokrytí cest. Cílem této práce bylo tedy vytvořit doplněk do prostředí Eclipse, podporující tuto metriku.

V testování se využívá mnoho technik, které definují, jak vytvářet jednotlivé testovací případy, s cílem pokrýt kritické části kódu, nebo nejlépe kód celý. Metriky pak můžou vyjádřit úspěšnost daného pokrytí. První kapitola je tedy věnována testování softwaru, používaným technikám a jednotlivým metrikám.

Při vývoji nástroje do prostředí Eclipse využívající metriku pokrytí cest je použit prototyp nástroje[1], který danou metriku již využívá. Současně je také využíván doplněk do Eclipse EclEmma [5]. Ten slouží spíše jako inspirace k vytvoření funkčního nástroje do daného prostředí. Oba tyto nástroje jsou pak popsány v další kapitole, spolu s jejich funkčnosti a možnosti jejich rozšíření. Dále pak následuje samostatná kapitola zaměřená na zmíněný prototyp nástroje, Z důvodu jeho dalšího využití, je podrobněji popsán jeho stav a princip fungování důležitých části.

Využívaný prototyp byl následně rozdělen na dva moduly. Ty jsou upraveny a přizpůsobeny pro správný chod v prostředí Eclipse. První z nich je původní prototyp, implementující metriku pokrytí cest. Druhý pak modul nový, obsahující prvky uživatelského rozhraní a prezentující výsledky v prostředí Eclipse. Těmto úpravám je věnována další kapitola.

Po všech úpravách a optimalizacích bylo potřeba jednotlivé moduly připravit na další distribuci. To obnáší jejich sloučení do jednotného celku, s cílem vytvoření výsledného nástroje nesoucí jméno CodeCoverage. K tomuto kroku se využívá nástrojů Maven[8] a Tycho[7]. Jak tyto nástroje fungují a následně vytváří výsledný nástroj, je obsahem další kapitoly.

Po kompletním sestavení nástroje CodeCoverage z původních modulů, je potřeba zařídit jeho distribuci. Této problematice je věnována následující kapitola. Obsahuje zejména popis, jak je nástroj uvolňován na webový server a jakou strukturu zde vytváří.

Poslední kapitola je pak věnována popisu celého procesu sestavování a uvolňování nástroje CodeCoverage.

2 Úvod do testování softwaru

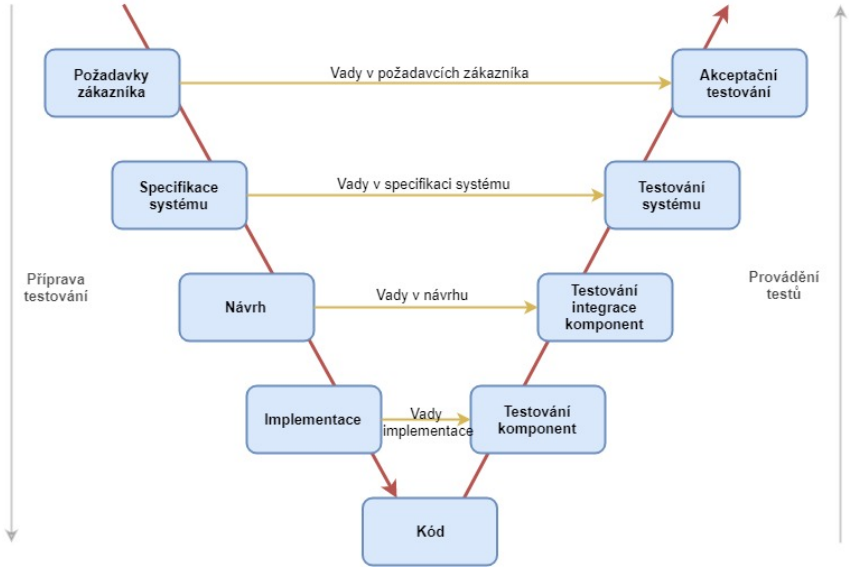
Testování softwaru je proces ověřování, zda jsou splněny specifikace softwaru zadané klientem [3]. Z postupem času se tato disciplína rozrůstá, stejně jako další disciplíny softwarového inženýrství. Nyní tvoří testování nezbytnou aktivitu při vývoji SW.

Hlavní tři činnosti prováděné při testování SW jsou:

1. **Verifikace** - proces ověřování, jestli je systém vytvářen správně.
2. **Detekce chyb** - cílem této činnosti je úmyslně vyvolat chyby, a sledovat jestli systém reaguje tak jak by měl.
3. **Validace** - proces ověřování správnosti výrobku s ohledem na potřeby a požadavky zákazníka.

2.1 Testování při vývoji softwaru

Testování už v průběhu vývoje softwaru má svůj účel. Čím dřív totiž odhalíme chybu ve vyvíjeném softwaru, tím dřív ji můžeme opravit a tím méně nás to bude stát. Tento postup testování má předejít vysokým nákladům, vynaloženým k případné opravě chyb v již skoro dokončeném, či hotovém softwaru. Pro testování v průběhu vývoje se používá tzv. V-model znázorněn na Obr. 1.



Obrázek 1: V-model

Pro každou fází v rámci vývoje software existuje fáze testování.

- Testování komponent - nebo také Unit testování. První fáze testování, která se zaměřuje na nejmenší části aplikace - moduly, třídy a metody.

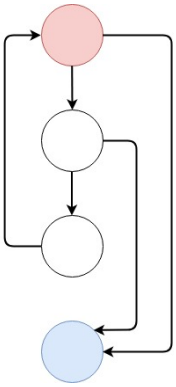
- Testování integrace komponent - se zaměřuje na testování korektní komunikace jednotlivých modulů resp. aplikací. Může být rozdělená na vnitřní (komunikace mezi moduly dané aplikace) a vnější (propojování aplikací do větších celků).
- Testování systému - jde o ověření, že aplikace jako celek funguje správně.
- Akceptační testování - ověřování že aplikace splňuje všechny zákazníkovy požadavky

2.2 Techniky testování

Existují dva základní soubory technik využívané při testování. Jedná se o Black-box a White-box. Black-box techniky jsou zaměřeny na testování funkcionality jednotlivých testovacích objektů. Známe tedy pouze vstupy a výstupy testovacího případu, které získáme z požadavků či návrhu, a implementace je pro nás neznámá. Naopak u White-box technik vycházíme z implementace dané funkcionality. Po analýze konkrétního kódu dokážeme sami určit co bude vstupem testu. Testování je založeno na logice programu.

2.2.1 Cyklomatická složitost

je softwarová metrika, používaná v rámci jednotlivých technik, vyjadřující míru složitosti programu. Měří počet lineárně nezávislých cest přes zdrojový kód programu. Jedná se sice o metriku používanou při statické analýze kódu, je ale úzce spjata s white-box technikami pro měření pokrytí kódu. Cyklomatická složitost se vypočítá za pomocí grafu řízení toku(dále CFG) znázorněného na Obr.2.



Obrázek 2: CFG části zdrojového kódu.

Tento graf znázorňuje smyčku s dvěma ukončeními, např. *while* s *if...break* podmínkou uprostřed. Výsledná složitost se pak vypočítá podle vzorce:

$$M = E - N + 2P.$$

- **M** = cyklomatická složitost

- **E** = počet hran v grafu
- **N** = počet uzlů v grafu
- **P** = počet připojených komponent

Dosadíme-li do vzorce podle našeho ukázkového grafu, vyjde nám následující složitost:

$$M = 5 - 4 + 2 * 1$$

$$M = 3$$

Toto číslo nám pak následně může pomoct stanovit počet testovacích případů nezbytných k důkladnému otestování konkrétního modulu. Toto je užitečné hlavně díky dvou vlastnostem cyklomatické složitosti M pro daný modul.

1. M tvoří horní hranici počtu testovacích případů, které jsou nezbytné k dosažení kompletního pokrytí pomocí testování skoků/rozhodování.
2. M tvoří spodní hranici počtu možných cest, které lze dosáhnout v CFG.

Všechny tři zmíněná čísla se ale mohou rovnat:

$$testování\ skoků/rozhodování \leq cyklomatická\ složitost \leq počet\ cest$$

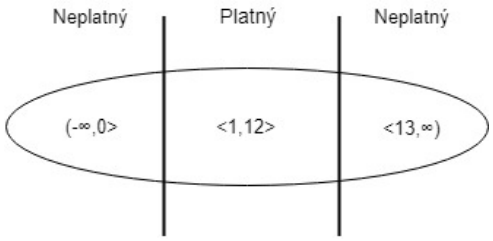
2.2.2 Black-box techniky

Black-box techniky slouží k testování funkcionalit, u kterých předem neznáme jejich implementaci. Je potřeba se spolehnout pouze na zadané vstupy a očekávané výstupy. Dále jsou popsány nejpoužívanější black-box techniky.

Ekvivalentní rozdělení: Identifikují se množiny vstupních hodnot, za předpokladu, že testovaný systém zachází stejně se všemi hodnotami v dané množině. Pro každou identifikovanou množinu se vytvoří alespoň jeden testovací případ. Jedná se o nejzákladnější testovací techniku. Mějme funkci, která má jeden vstupní parametr "měsíc". Platná vstupní množina je tedy 1 až 12, reprezentující měsíce od ledna po prosinec. Existují tady ale ještě dvě další množiny. První je <= 0 a druhá je >= 13. Tyto množiny nazýváme neplatné. Viz. příklad na Obr. 3.

Analýza hraničních hodnot: Pro každou identifikovanou hranici ve vstupech a výstupech se vytvoří dva testovací případy. Jeden na každé straně hranice, tak blízko jak jen to je možné.

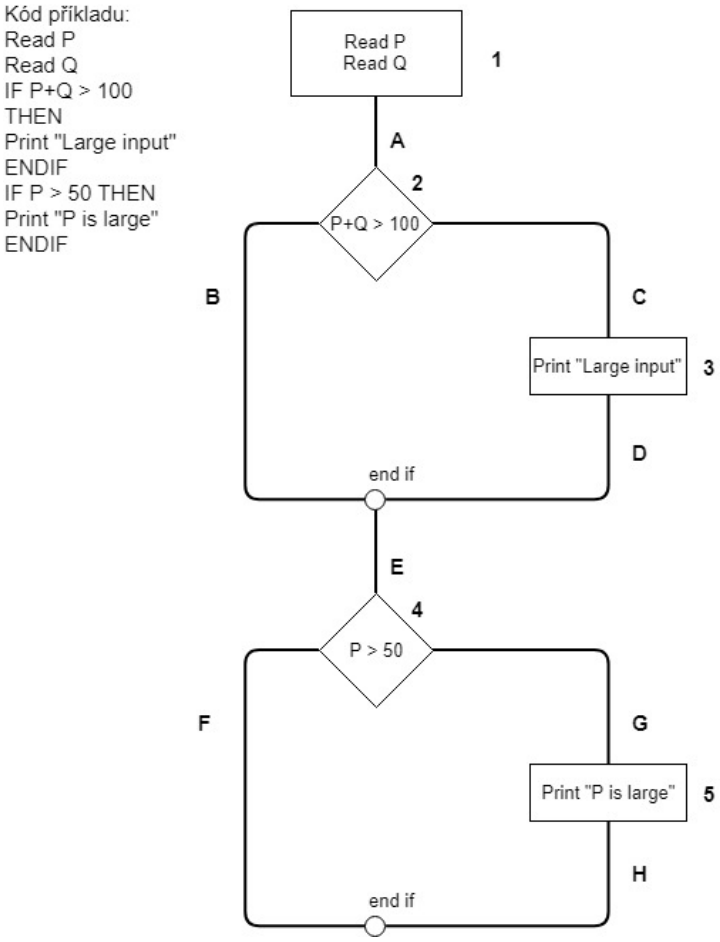
Testování přechodu mezi stavy: Funkční chování je namodelováno pomocí stavového automatu. Používá se v embedded systémech.



Obrázek 3: Ukázka ekvivalentního rozdělení

2.2.3 White-box techniky

Následující tři techniky testování představím na příkladu. Obr. 4



Obrázek 4: Příklad pro demonstraci technik testování

Testování výrazů/instrukcí (SC) je takové testování, kde je potřeba vytvořit tolik testovacích případů, aby bylo zaručeno, že se každá instrukce provede alespoň jednou.

K výpočtu pomocí metriky pokrytí výrazů/instrukcí, potřebujeme zjistit nejmenší počet cest,

kterými pokryjeme všechny uzly(instrukce). Když se podíváme na příklad na Obr. 4 zjistíme, že pokud projdeme cestou *1A-2C-3D-4G-5H*, pokryjeme tak všechny možné uzly. Takže průchodem pouze jedné cesty, jsou všechny naše uzly(instrukce) *12345* pokryty. Výsledek pokrytí podle této metriky na tomto příkladu je 1.

Testování skoků/rozhodování (BC) je založeno na snaze pokrýt všechny rozhodování, které v kódu nastanou. Každé rozhodování či skok v kódu, je třeba pokrýt s obou stran, tedy *true* a *false*. Tato metoda pokrytí kódu nám pomůže ověřit, že žádná větev námi procházených rozhodnutí, nevede k abnormálnímu chování aplikace.

K výpočtu pomocí metriky skoků/rozhodování, potřebujeme zjistit minimální počet cest, kterými pokryjeme všechny hrany(větve). Vycházíme-li z příkladu z předchozí stránky, vidíme že zde neexistuje žádná cesta, která by zajistila pokrytí všech větví najednou. Průchodem cesty *1A-2C-3D-E-4G-5H* si zajistíme pokrytí většiny větví(A, C, D, E, G, H). Zůstanou nám větve B a F. K pokrytí těchto větví můžeme zvolit cestu *1A-2B-E-4F*. Kombinací těchto dvou cest jsme schopní zajistit pokrytí všech možných větví. Výsledek pokrytí pomocí dané metriky je 2. Cílem je pokrýt všechny *true*/*false* rozhodnutí.

Testování pokrytí cest (PC) se zaměřuje na průchod všemi možnými cestami alespoň jednou. Tyto cesty zahrnují také průchod smyčkami a to alespoň nula, jednou a více krát(nejlépe však maximum). Z těchto třech zmíněných technik se jedná o nejsložitější. K výpočtu dané metriky se používá Cyklomatická složitost, CFG, a metriky používané v grafech. Někdy je skoro nemožné nalézt všechny cesty pro daný případ. První problém se týká smyček, které mohou být nekonečné. Další problém jsou neuskutečnitelné cesty, a tedy cesty, ke kterým při testování neexistuje žádný vstup, který by zaručil průchod danou cestou. Pokrytí cest nám zajišťuje průchod všemi cestami od začátku až do konce. Všechny tyto cesty jsou:

- *1A-2B-E-4F*
- *1A-2B-E-4G-5H*
- *1A-2C-3D-E-4G-5H*
- *1A-2C-3D-E-4F*

Metrika pokrytí cest je nejsložitější na výpočet. U jednoduchých příkladu jako ten zmíněný výše, je výpočet jednoduchý od pouhého pohledu. Existují ale složitější funkce či algoritmy, ve kterých je těžké se zorientovat, natož určit počet všech možných cest. Tento nástroj by měl tuto práci ulehčit. Jeho zaměření je hlavně na „testery“, kterým pomůže při hledání možných cest v kódu, a ulehčí tak psaní jednotlivých testů.

3 Analýza aktuálních nástrojů a možnosti jejich rozšíření

V současné době existuje mnoho nástrojů v oblasti testování pro měření pokrytí kódu. Každé řešení se vztahuje k určitému programovacímu jazyku a konkrétnímu IDE. V této kapitole budou představeny dva nástroje, které byly použity při vývoji výsledné aplikace.

3.1 CodeCoverage

CodeCoverage [1] je prototyp nástroje pro Eclipse, který dokáže analyzovat kód pomocí metriky pokrytí cest. Tento nástroj byl vyvíjen v rámci diplomové práce a byl zpřístupněn z myšlenkou dokončení do použitelné a instalovatelné podoby.

3.1.1 Funkce nástroje

Jak již bylo zmíněno jedná se o prototyp nástroje pro prostředí Eclipse. Dokáže změřit jednotlivé cesty průchodu kódu a ty následně vykreslit do editoru pomocí anotací Obr.5. Nástroj podporuje pouze programovací jazyk Java. K měření využívá knihovnu JaCoCo, která sbírá data o jednotlivých řádcích kódu. Tyto data následně analyzuje a zobrazí informace o průchodech jednotlivých metod. Spuštění celé analýzy probíhá pomocí JUnit frameworku. Lze tedy spustit

```
@Test
public void test10() {
    method(10);
    method(9);
}

public void method(int a) {
    System.out.println("Hi");
    if (a == 10) {
        System.out.println("Hi");
    } else {
        System.out.println("Hi");
    }
}
```

Obrázek 5: Vykreslení výsledné cesty v editoru

pouze projekt obsahující anotace jako *@Test*, které značí použití tohoto frameworku.

3.1.2 Architektura

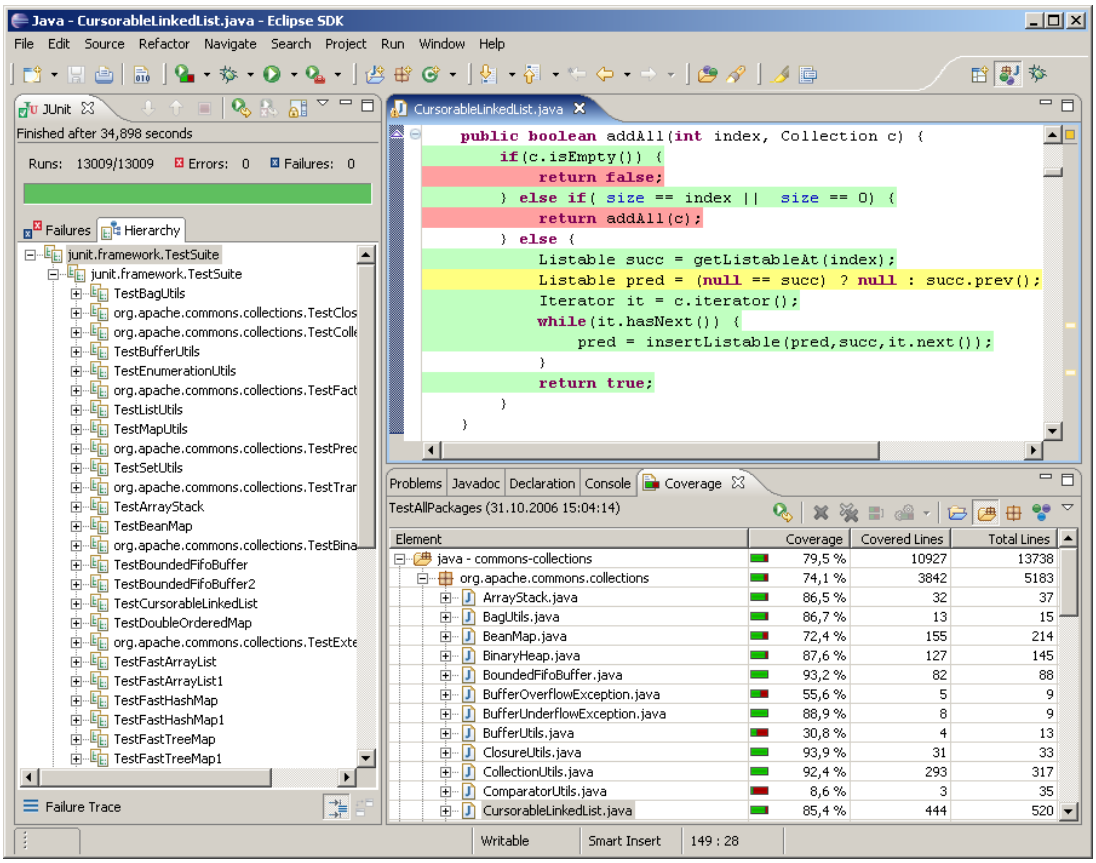
CodeCoverage je tvořen samostatným modulem(pluginem) do prostředí Eclipse. Tento plugin obsahuje jak veškerou programovou logiku, tak uživatelské rozhraní. Dále je využívána modifikovaná knihovna JaCoCo - knihovna pro pokrytí kódu v jazyce Java. Ta pak ještě interně využívá další knihovnu ASM, která slouží s manipulováním Java bytecode. Více o tomto nástroji v kapitole 4.

3.1.3 Možnosti rozšíření

Rozšíření tohoto nástroje o další funkčnost, či případné úpravy stávajících vlastností, jsou možné. Je to prototyp, a tedy první pokus o vytvoření kompletního nástroje pro prostředí Eclipse. Jelikož tento nástroj není komplexní a neobsahuje mnoho funkčností, je možné tento nástroj snadno rozšířit. Jedinou větší překážkou je tak neznalost použitých technologií, zvlášť PDE.

3.2 EclEmma

EclEmma [5] je volně dostupný nástroj pro měření pokrytí kódu v jazyce Java. Poskytuje analýzu kódu pomocí metriky skoků/rozhodování a výsledek zobrazí přímo do editoru. Využívá k tomu knihovnu JaCoCo. Obsahuje mnoha módu spuštění dostupných přímo z pracovního prostředí Eclipse. Dokáže také importovat předešlé analýzy, nebo exportovat do jiných formátů jako jsou HTML, XML nebo CSV. Ukázka výsledné analýzy je vyobrazena na Obr. 6



Obrázek 6: Ukázka analýzy nástroje EclEmma v prostředí Eclipse. Zdroj: [5]

3.2.1 Funkce nástroje

Jednou z hlavních vlastnosti nástroje je možnost spuštění požadované aplikace s cílem získání dat o pokrytí. EclEmma to nazývá mód pokrytí(*Coverage* mode) a funguje naprosto stejně jak módy

pro běžné spouštění aplikací v Eclipse - *Run* a *Debug*. Jednoduché spuštění aplikace v módu pokrytí nám zajistí získání informací o pokrytí. V současné době jsou podporovány následující typy spouštění:

- Local Java application
- Eclipse/RCP application
- Equinox OSGi framework
- JUnit test
- TestNG test
- JUnit plug-in test
- JUnit RAP test
- SWTBot test
- Scala application

Získané informace o pokrytí po ukončení běhu dané aplikace jsou pak automaticky dostupné v prostředí Eclipse. Obecné informace o pokrytí lze nalézt v tzv. „Coverage view“. Je to okno s informacemi o pokrytí pro Java projekty, zobrazující detaily až na úroveň metod. Coverage view nám pak dále poskytuje informace o počtu pokrytých instrukcí, rozhodování, řádků kódu, metod a dokonce cyklomatickou složitost.

Další možnost zobrazení těchto informací je přímo v Java editoru. Výsledek je automaticky viditelný, hned po ukončení běhu v módu pokrytí. V editoru se pak zabarví jednotlivé řádky kódu podle toho jestli byly nebo nebyly pokryty, případně částečně pokryty. Zobrazení v editoru je nastavitelné a ve výchozí konfiguraci se zobrazují barvy zelená, červená a žlutá. Zvýrazní se jak zdrojový kód spuštěného projektu, tak používaných externích knihoven. Nástroj umožní také přepínat mezi daty o pokrytí z mnoha spuštění.

EclEmma je primárně navržená na analýzu přímo v prostředí Eclipse, umožňuje ale taky import a export těchto informací.

- **Import** - Nástroj umožňuje importovat JaCoCo *.exec soubor, který obsahuje veškeré informace o pokrytí pro konkrétní spuštění.
- **Export** - Data o pokrytí mohou být exportovány do formátu jako HTML, XML nebo CSV, nebo jako JaCoCo spouštěcí datový soubor(*.exec).

3.2.2 Architektura

EclEmma je tvořena ze dvou modulů. První „core“ modul obstarává všechnu funkcionalitu pro spouštění a analýzu. Nemá žádné závislosti na uživatelské rozhraní Eclipse. Poskytuje veřejné API, pomocí kterého lze s tímto modulem komunikovat a používat jeho funkce. Druhý „UI“ modul poskytuje integraci do pracovního prostředí Eclipse. Z druhým modulem komunikuje přes již zmíněné API „core“ modulu.
Klíčové vlastnosti podle kterých je EclEmma navržena a implementována:

- Vysoká použitelnost umožňující vývojářům efektivně analyzovat výsledky pokrytí a zlepšit jednotlivé testy interaktivně.
- Bezproblémová integrace do pracovního prostředí Eclipse, použitím stávajících nástrojů a paradigmat.
- Primární cíl je spouštění a analýza přímo z pracovního prostředí Eclipse. Pro automatizované sestavení a tvoření reportů JaCoCo umožňuje integraci z nástroji jak Ant či Maven.
- V průběhu analýzy čerpá informace z projektů ale nijak jich nemodifikuje. Proto tento nástroj pracuje bez zasažení do stromové struktury projektu či jeho konfigurace.

3.2.3 Možnosti rozšíření

Jak už bylo zmíněno výše, jeden z modulů EclEmma poskytuje API ke svým funkcím. Toto API umožňuje přístup ke spouštění a analýze daného projektu. Je tak možné vytvořit své vlastní uživatelské rozhraní nebo případně vylepšit či změnit to aktuální. Upravit a rozšířit jde samozřejmě i „core“ modul. K jeho úpravě je ale potřeba mít dobré znalosti o Eclipse launch a debug frameworku. Tento nástroj je vyvíjen už více než 10 let a je poměrně komplexní. K úpravě tak složitého nástroje je dobré mít znalosti z oblasti vývoje doplňků do prostředí Eclipse a celkově znalost PDE.

3.3 Souhrn obou nástrojů a cílové řešení

Oba výše představené nástroje se zabývají problematikou pokrytí kódu a obsahují podobnou funkcionalitu. Oba nástroje je také možno rozšířit. CodeCoverage má výhodu že dokáže použít metriku pokrytí cest. EclEmma sice používá jinou metriku ale tento nástroj je kompletně funkční a sestavený a je dostupný ke stažení a instalaci z Eclipse Marketplace [2]. Nabízejí se tedy tři možnosti jak vytvořit hotový nástroj pro Eclipse, schopný pokrytí kódu pomocí metriky pokrytí cest.

1. Rozšíření nástroje EclEmma o vlastní implementaci metriky pokrytí cest
2. Spojení obou nástrojů dohromady
3. Rozšířit, upravit a dokončit nástroj CodeCoverage

3.3.1 Rozšíření EclEmma

Rozšíření nástroje EclEmma o vlastní implementaci metriky pokrytí cest bylo zavrženo hned na začátku. Vzhledem ke složitosti tohoto nástroje a neznalosti v oblasti PDE, by trvalo dlouhou dobu pochopit jeho implementaci. Samotný návrh a implementace druhé metriky by pak taky zabral mnoho času.

3.3.2 Spojení obou nástrojů

Tato možnost se zdá být trochu reálnější. Je zde ale stále problém se složitosti nástroje EclEmma. Oba nástroje používají ke svému chodu knihovnu JaCoCo. CodeCoverage ale používá vlastní modifikací této knihovny. Použití dvou verzí knihovny, z čehož ta modifikována není oficiálně distribuována a licencována, je taky zásadní problém. Obecně je prostředí Eclipse a veškerá jeho funkcionalita tvořena mnoha doplňky. Je tedy logičtější poskytovat každou novou funkčnost v samostatném nástroji, místo spojování dvou existujících.

3.3.3 Rozšíření CodeCoverage

Volba rozšířit nástroj CodeCoverage je ze všech možnosti nejreálnější. Tento nástroj obsahuje pokrytí pomocí metriky pokrytí cest. Je třeba pouze nástroj důkladně pochopit, opravit některé chyby, upravit uživatelské rozhraní a umožnit sestavení a distribuci.

3.3.4 Cílové řešení

Z důvodů výše popsaných bylo zvoleno řešení rozšířit stávající nástroj CodeCoverage. EclEmma bude sloužit jako příklad pro různé oblasti vývoje doplňků pro Eclipse.

4 Detailní popis stávajícího modulu

V minulé kapitole byly představeny různé možnosti vytvoření výsledného nástroje a taky představeno konkrétní řešení. Na základě tohoto rozhodnutí, bude v této kapitole podrobněji popsán nástroj CodeCoverage. Tento nástroj bude následně upravován a proto je potřeba se s nim detailněji seznámit.

Táto kapitola bude rozdělena na tří části. Jedna se bude věnovat knihovně JaCoCo, její úpravám a propojení s modulem. Další část bude popisovat analýzu kódu a vytvoření jednotlivých průchodů/cest. V poslední části pak bude popsán způsob prezentace získaných výsledku v prostředí Eclipse.

4.1 Knihovna JaCoCo a její úpravy

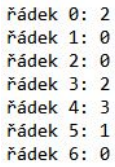
Tento nástroj používá pro analýzu kódu knihovnu JaCoCo(Java Code Coverage Library). Tato knihovna je využívána prakticky všemi nástroji, které se zabývají pokrytím kódu v jazyce Java. JaCoCo přímo upravuje Java bytecode a zpětně pak získává data, která indikují jestli se dané instrukce provedly, či nikoli. Knihovna pak dokáže převést tyto data na jednotlivé řádky kódu. Tyto informace už jsou pak využívány jednotlivými moduly.

4.1.1 Princip fungování JaCoCo

JaCoCo funguje na principu vkládání vlastního kódu do používaného programu s informací, jestli se daná instrukci, či posloupnost instrukcí, provedla. Knihovna neupravuje samotný Java kód, ale Java bytecode. Ten vzniká po zkompilování Java kódu a je následně používán JVM pro spuštění dané aplikace. K úpravě Java bytecode využívá JaCoCo framework ASM. Díky tomu pak může vkládat vlastní kód s informací, které instrukce se provedly. Tento vložený kód(tzv. „probe“) uloží *true* pokud se daný úsek kódu provedl. JaCoCo pak dbá na to, aby vložený kód nijak neovlivnil kód původní. Určuje také jakým způsobem se „probe“ do kódu vkládají. Pokud by se totiž vkládaly po každé instrukci zvlášť, mohlo by to vést ke značnému zpomalení programu. Výsledek pokrytí kódu se pak nachází v poli typu *boolean*. Analýza kódu pouze pomocí tohoto pole je ale velmi pracná. Naštěstí JaCoCo poskytuje funkci, která tyto *boolean* hodnoty převede na jednotlivé řádky. Výsledek pak může nabývat těchto hodnot:

- **0** - Daný řádek neobsahuje kód
- **1** - Kód daného řádku se neprovedl
- **2** - Kód se provedl. V případě větvení i všechny možnosti
- **3** - U větvení se provedly jen některé možnosti

Samotný výstup po analýze daného kódu lze vidět na Obr. 7. Kromě samotného pokrytí



Obrázek 7: Výsledek analýzy kódu

jednotlivých řádku, jsou dostupné i počty nepokrytých instrukcí, větví či řádků.

4.1.2 Úprava JaCoCo

JaCoCo tedy poskytuje informace o pokrytí na daném kódu. Z těmito informacemi je možné snadno vytvořit statistiku o pokrytí pomocí metriky pokrytí rozhodování/skoků. Jednotlivé cesty průchodů už nelze zjistit. Z tohoto důvodu je knihovna patřičně upravena.

K tomuto účelu byl vymyšlen princip časových značek(tzv. „timestamp“). Ukládáním těchto časových značek k jednotlivým „probe“, bude následně možno analyzovat jak se daný kód provedl. Je k tomu používána funkce *nanoTime* ze třídy *System*. Tato funkce vrací aktuální hodnotu časového zdroje běžícího JVM. Pokud si uložíme přesný čas kdy se daný kód provedl, můžeme pak zjistit jak se provedl.

Ke každému probe je potřeba uložit časové značky. Jelikož program může(a často se tak děje) jednotlivé instrukce či části kódu procházet více krát, tím pádem bude i více krát procházet vložené probe. K tomuto účelu je nutno ke každé probe uložit kolekci časových značek. Jelikož kód obsahuje více probe, musí se pro každou probe uložit vlastní časové značky. K tomu se využívá hashovací tabulka, kde klíč je číslo probe a hodnota je kolekce časových značek. Výsledný atribut pro ukládání časových značek vypadá takto: `Map<Integer,List> timeStampData`.

Úprava samotné knihovny JaCoCo se pak týká všech tříd a metod, které pracují se zmíněnými probe. Tyto změny se týkaly přibližně 26-ti tříd. V některých byly úpravy minimální a v dalších bylo zasaženo do implementace více. Mezi hlavní upravované třídy patří např. *ProbeInserter*, kde probíhá inicializace proměnných pro vkládání bytecode či *ExecutionData*, která shromažďuje výsledná data.

4.1.3 Propojení JaCoCo s modulem

Samotnou úpravu Java bytecode a následné předání výsledných dat spolupracujícímu modulu obstarává tzv. „Java agent“. Je to vlastně modul do JVM, který se spouští před vykonáním samotného programu. Ten může pozměnit class soubory, které se před spuštěním programu nahrají do JVM. Po ukončení běhu programu pak výsledná data dokáže zaslat různými způsoby podle toho, jak ho nakonfigurujeme. V JaCoCo se používaný agent nazývá jacocoagent. Pro spuštění programu s java agentem, se zadá parametr ve tvaru *-javaagent:cesta/k/jacocoagent.jar*. Za tímto

parametrem může následovat až 14 dalších parametrů *...jacocoagent.jar=[param1]=[value1], [param2]=[value2]*. Těmito parametry můžeme nakonfigurovat jak se má agent spustit. Nejdůležitější parametry jsou:

- **destfile** - Cesta k uložení výstupního *.exec souboru.
- **output** - Způsob uložení dat.
  - **file** - Data se uloží jako soubor. Cesta je v *destfile*.
  - **tcpserver** - Agent vytvoří TCP server. Klient se na něj připojí a příjme data
  - **tcpclient** - Agent se přípojí k serveru.
  - **none** - Agent nemá vytvořit žádný výstupní soubor.
- **address** - IP adresa nebo „hostname“ serveru pro připojení.
- **port** - Port na který se má připojit či naslouchat.

Modul CodeCoverage potom používá následnou konfiguraci spuštění:

-javaagent:cesta/k/jacocoagent.jar=output=tcpclient,address=localhost,port=9999

V modulu pak existuje třída, která spustí server a čeká na připojení klienta(jacocoagent).

4.2 Analýza kódu a vyhodnocení cest

Celý proces analýzy a vyhodnocení pokrytí je rozdělen na dvě části. Při analýze kódu se vytvoří stromová struktura spouštěného programu. To umožňuje efektivnější práci při následném vyhodnocení pokrytí cest. Následné vyhodnocení už pak jednoduše řečeno, proběhne „průchodem“ vytvořeného stromu.

4.2.1 Vytvoření struktury kódu

K vytvoření stromové struktury kódu se využívá třída AST, která je součástí JDT. Samotný název třídy AST(Abstract Syntax Tree) naznačuje že se jedná o abstraktní syntaktický strom. Toto označení představuje v informatice stromovou reprezentaci abstraktní syntaktické struktury zdrojového kódu. AST je podobný k DOM, který poskytuje rozhraní k manipulaci s HTML či XML. Nástroj CodeCoverage pak tuto třídu využívá k „rozparsování“ potřebného kódu. Pro vytvoření výsledné stromové struktury, pak využívá vlastní implementaci *ASTVIsitor*.

K vytvoření této struktury se nejprve používá třída *ASTParser*. Ta dokáže vytvořit syntaktický strom kódu z obyčejného textového řetězce. Vstupním parametrem je dokument, který chceme převést, výstupem pak AST struktura. To ale není vše. Pro finální část analýzy se používá vlastní stromová struktura. K tomuto účelu jsou v nástroji třídy, reprezentující jednotlivé instrukce. Tyto třídy tvoří dvě skupiny. První skupina reprezentuje jednoduché instrukce(volání metody, nová instance třídy, aritmetické operace atd.) a jsou reprezentovány následujícími třídami:

- AbstractSimpleCodeElement
- SimpleCodeElement
- LabelCodeElement
- SpecialSimpleCodeElement
- BreakCodeElement
- ThrowCodeElement
- ReturnCodeElement
- ContinueCodeElement

Druhá skupina reprezentuje instrukce, které větví program(např. *if*, *for*, *do-while*):

- AbstractBranchCodeElement
- MethodElement
- WhileCodeElement
- DoWhileCodeElement
- ForCodeElement
- SwitchCaseCodeElement
- SwitchCodeElement
- TryCodeElement
- IfCodeElement
- ForeachCodeElement
- CatchCodeElement
- BlockCodeElement

Dalším krokem k vytvoření vlastní stromové struktury je použití třídy *ASTVisitor*. Ta využívá návrhový vzor *Návštěvník*. Zde existuje pro každý operátor a operand metoda *visit*. K tomuto účelu se využívá přetěžování metod(stejné metody s jinými parametry). Hlavička metody *visit* pro navštívení podmínky *if* vypadá takto: *visit(IfStatement node)*.

Tento modul obsahuje vlastní implementaci *ASTVisitor*. Díky tomu dokáže rozšířit potřebné metody *visit* o vlastní implementaci. V těchto metodách se inicializují jednotlivé třídy reprezentující instrukce. Jednotlivým třídám se pak nastavují tři základní vlastnosti. Začátek a konec řádku a kolekce časových značek.

4.2.2 Vyhodnocení pokrytí cest

Vyhodnocení pokrytí cest probíhá v metodě *analyze*. Tuto metodu implementují všechny výše zmíněné třídy reprezentující instrukce. Metoda obsahuje tři parametry a její hlavička vypadá takto:

```
public AnalyzeReturn analyze(Method method, ICodeElement nextElement, long
                             nextIterationTS)
```

První parametr je reference na třídu *Method*. Do této třídy se ukládají veškeré výsledky. Druhý je reference na instrukci, která následuje v předešlém bloku a třetí je časová značka následujícího průběhu metody. Metoda *analyze* má samozřejmě odlišnou implementaci vzhledem k jednotlivým instrukcím. Celý průchod stromové struktury kódu je pak řízen díky návratovému typu této metody. Ten obsahuje informaci o tom jak se daný příkaz/blok provedl. Existuje osm návratových typů:

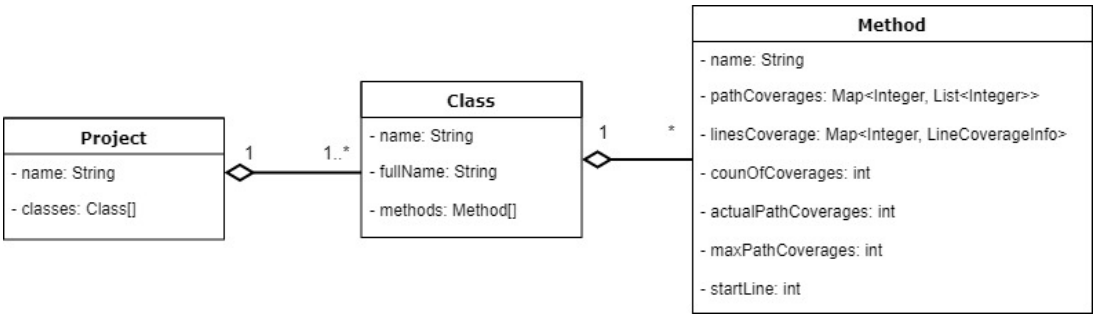
- **return** - Analyzátor narazil na příkaz *return*
- **break** - Analyzátor narazil na příkaz *break*

- **continue** - Analyzátor narazil na příkaz *continue*
- **throw** - Analyzátor narazil na příkaz *throw*
- **not_this_block** - V případě že se daný blok neprovedl, ale má se pokračovat dál
- **no_such_timestamp** - Příkaz či blok neobsahují další časové značky
- **ok** - Příkaz či blok se provedly bez problému
- **throws** - Nastala vyjímka

Celý princip funguje na procházení jednotlivých příkazů jak jdou za sebou a porovnávání časových značek. V případě že se daná instrukce provedla v pořádku, uloží se číslo řádku do příslušné struktury a časová značka se označí jako použitá.

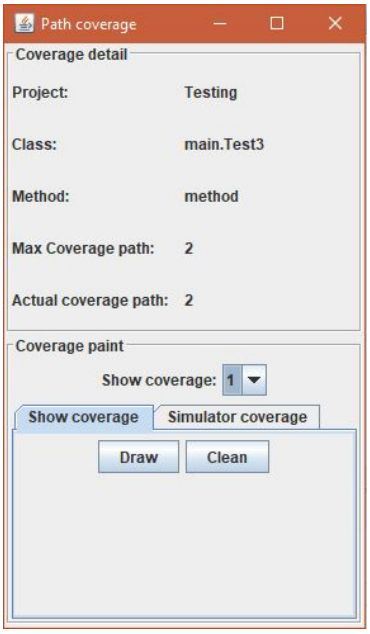
4.3 Zobrazení pokrytí cest

K zobrazení získaných informací o pokrytí cest se využívají(až na jednu výjimku) komponenty používané v pracovním prostředí Eclipse. Požadované data se nacházejí ve třech vzájemně propojených třídách *Project-Class-Method* reprezentující analyzovaný program. Tyto třídy obsahují následující vlastnosti(viz. Obr. 8). Jsou pak využity pro zobrazení stromové struktury obsahující



Obrázek 8: Třídní diagram tříd obsahující informace o pokrytí

projekty, třídy a metody. Tato struktura je zobrazena v tzv. „view“. Uživatelské rozhraní Eclipse je složeno z několika takových view, které tvoří obdélníkové oblasti zobrazující obsah. Mezi známe view v prostředí Eclipse patří např. *Console*, *Outline* nebo *Package Explorer*. Jednotlivé informace o pokrytí se pak zobrazí po dvojkliku na příslušnou metodu. Zobrazí se samostatné okna typu *JFrame*, kde jsou vidět jednotlivé statistiky. Je zde možnost vybrat si příslušnou „cestu“(pokud pro danou metodu nějaká existuje) a následně ji vykreslit. Ukázka okna s pokrytím dané metody je na Obr. 9



Obrázek 9: Okno s informacemi o pokrytí metody

Jednotlivé cesty jsou pak vykresleny přímo do editoru pomocí anotací. Vidíme tak výsledek přímo na kódu. Ukázka zobrazení pomocí anotací je v kapitole 3.1.1 na obrázku 5.

Poslední respektive první věcí je spuštění samotné analýzy. K tomuto účelu slouží tlačítko v *toolbaru*. *Toolbar* je v pracovním prostředí Eclipse lišta z různými tlačítky ať už platformy Eclipse nebo jiných instalovaných modulů. Celkové spuštění analýzy probíhá tak, že se daný projekt spustí jako JUnit(framework pro jednotkové testy) projekt. Po skončení běhu se spustí samotná analýza, která následně inicializuje potřebné třídy a naplní je jednotlivými daty. Tyto třídy už pak využívá view pro svoje účely.
Tlačítko v *toolbaru* má tři možnosti spuštění:

1. RunJUnit - Projekt se spustí jako JUnit
2. RunJUnitLast - Spustí se posledně spuštěný projekt znovu jako JUnit
3. Analysis - Mimo spuštění JUnit a analýzy, ještě vypíše do konzole detailní informace o průběhu vyhodnocování. Bylo používáno hlavně při vývoji.

5 Rozšíření a úpravy dosavadního modulu

Po prozkoumání několika možností bylo cílovým řešením této diplomové práce zvoleno rozšíření rozpracovaného nástroje CodeCoverage(viz. kapitola 3). Aktuální stav toho nástroje je popsán v předešlé kapitole. Tento nástroj byl rozpracovaný a obsahoval několik nedostatků, které se týkaly hlavně chování modulu v prostředí Eclipse a jeho ovládání. V této kapitole budou tedy popsány úpravy, které byly provedeny. Z velké části tyto úpravy souvisely s optimalizací tohoto nástroje pro správný chod v prostředí Eclipse [4].

5.1 Rozdělení modulu

Vytvoření jedné aplikace, většinou neznamená vytvoření pouze jednoho projektu či programu. V dnešní době většina vývojářů používá pro vytvoření svoji aplikace některou z definovaných architektur(např. MVC, Klient-Server ...). Tyto architektury definují vrstvy, do kterých by měla být aplikace rozdělená. Každá vrstva je pak zodpovědná za určitou funkčnost a jednotlivé vrstvy mezi sebou komunikují přes definována rozhraní. Toto rozdělení aplikace je zapotřebí hlavně v situacích, kdy každá vrstva běží na různé výpočetní infrastruktuře. Může ale také sloužit k logickému rozdělení aplikace.

Nástroj CodeCoverage je tvořen jedním modulem, který běží pouze v prostředí Eclipse. Tento modul obsahuje jak veškerou komunikaci s knihovnou JaCoCo, celou logiku analýzy, tak uživatelské rozhraní. Všechny tyto části se pak prolínají.

Většina nástrojů v prostředí Eclipse je tvořena několika moduly zároveň. Z těchto důvodů bylo rozhodnuto rozdělit aktuální modul na dvě části. Po dokončení všech úprav obsahují jednotlivé části následující funkčnost:

- **codecoverage.core** - Tento modul je prakticky modul původní. Jsou zde odebrány všechny prvky komunikující s uživatelským rozhraním. Modul obsahuje třídu zodpovědnou za komunikaci s druhým modulem. Má na starosti komunikaci s knihovnou JaCoCo, celkovou analýzu a uložení výsledku do potřebných tříd.

- **codecoverage.ui** - Jedná se o zcela nově vytvořený modul. Obsahuje uživatelské rozhraní, které zobrazuje výsledky analýzy v pracovním prostředí Eclipse. Komunikuje s *core* modulem. Využívá Eclipse 4 model.

Výsledkem je pak nástroj nesíci původní název CodeCoverage, obsahující dva výše zmíněné moduly.

5.2 Modul codecoverage.core

Tento modul je vlastně modul původní. Jeho název byl změněn z CodeCoverage na codecoverage.core, značící že obsahuje hlavní funkcionalitu tohoto nástroje. Bylo zde provedeno několik úprav, odstraněno a přidáno několik prvků.

5.2.1 Odstranění UI prvků

Na začátku bylo třeba z modulu odstranit veškeré prvky, které zasahovaly do uživatelského rozhraní Eclipse. O tuto funkčnost se stará druhý modul *codecoverage.ui*.

- *OwnView* - Třída, která se starala o zobrazení stromové struktury projektů. V druhém modulu je zcela nová a přepracovaná.
- *MyDialog* - Třída zobrazující okno s hláškou. Používalo se pro zobrazení různých informaci uživatelovi. V novém modulu je jiný způsob.
- *CoverageFrame* - Okno sloužící k vykreslení jednotlivých cest do editoru. Toto okno je v *UI* modulu nahrazeno zcela novým *view*. Technologie JFrame, která zde byla použitá, se v modulech pro Eclipse nepoužívá.
- *ButtonHandler* - *Handler*, který zde obstarával zpracování příkazu pro spuštění z uživatelského prostředí a samotné spuštění analýzy. V modulu *UI* je *handler* zcela nový a problematika spuštění je analýzy je přepracována v aktuálním modulu.
- *Annotations* - Třídy, které měly na starosti vykreslení pokrytí v editoru, byly po několika úpravách přesunuty do druhého modulu.

5.2.2 Úprava potřebných souborů

Úpravy se týkaly hlavně souboru *plugin.xml*. Tento soubor popisuje, jak modul rozšiřuje platformu, jaké rozšíření sám publikuje a jak implementuje její funkčnost. Jsou zde tedy definovány prvky uživatelského rozhraní jako jsou: *view*, ikony v liště, *handlery*. Všechny tyto definice bylo třeba odstranit. Další upravovaný soubor byla třída *ProjectParser*. Tato třída obsahovala mnoho funkcí, které se používaly při vykreslování do editoru. Obstarávala přístup k aktuálnímu dokumentu. To všechno díky neustálému přístupu k pracovnímu prostředí Eclipse pomocí třídy *PlatformUI*. Tento přístup není bezpečný a je vyřešený v druhém modulu pomocí asynchronních požadavků na platformu Eclipse.

5.2.3 Přidání nových souborů

Z důvodu nutných úprav modulu, bylo potřeba přidat několik nových tříd. Tyto třídy se starají o správný chod modulu a jeho komunikaci s druhým modulem. Všechny se nacházejí v balíčku *launching*.

Třída JUnitRunner: Tato třída má na starosti spouštění projektu jako JUnit a přípravu na následnou analýzu. Do metody vstupují dva hlavní parametry. První je reference na spouštěný Java projekt typu *IJavaProject* a druhý je cesta k samotnému projektu. Nejprve se najde aktivní otevřená třída pomocí upravené funkce v *ProjectParser*. Následně se spustí server pro sběr dat

od agenta. Poslední a nejdůležitější části je spuštění aktivního projektu jako JUnit. K tomu se využívá *Debug* framework prostředí Eclipse. Vytvoří se tzv. *ILaunchConfigurationWorkingCopy*, které se musí nastavit mnoho parametru pro správné spuštěni.

```
DebugPlugin.getDefault().getLaunchManager().getLaunchConfigurationType(
    JUnitLaunchConfigurationConstants.ID_JUNIT_APPLICATION);
wc.setAttribute(IJavaLaunchConfigurationConstants.ATTR_APPLET_PARAMETERS, "
    PathCoverage");
```

Výpis 1: Hlavní parametry *ILaunchConfigurationWorkingCopy*

První příkaz získá pomocí *DebugPlugin* konfiguraci pro spuštění projektu jako JUnit. Druhý pak nastaví speciální parametr pro identifikaci spouštěného programu. To je pak využíváno při registrování posluchače na událost ve třídě *PathCoverageActivator* 5.2.3. Samozřejmě je zde taky nastaven parametr, který přidává *jacocoagenta*. Následně zavoláme nad naší „*WorkingCopy*“ funkci *launch()* s parametrem *ILaunchManager*.RUN_MODE pro spuštění.

Třída PathCoverageActivator: Každý modul pro platformu Eclipse může obsahovat tzv. *Activator*, tedy něco jako spouštěcí třída. Tato třída ale nespouští samotný modul, nýbrž se stará o jeho celkový životní cyklus. Obsahuje dvě hlavní metody *start* a *stop*, které se volají při spuštění respektive ukončení modulu. Může tedy posloužit k inicializaci proměnných, které mají být k dispozici po celou dobu běhu modulu či registrování potřebných posluchačů na různé události. Po ukončení běhu modulu pak odstranění posluchačů a vynulování proměnných.

Aktivátor je obyčejná Java třída, která dědí s abstraktní třídy *AbstractUIPlugin* či *Plugin*. Pro registrování aktivátoru v našem modulu je potřeba přidat záznam do konfiguračního souboru MANIFEST.MF. Ten obsahuje závislosti modulu na jiných modulech, aktuální verzi či jméno. Mimo jiné také definici pro aktivátor.

Bundle-Activator: codecoverage.launching.PathCoverageActivator

V této třídě jsou inicializovány objekty, které mají na starosti spuštění a analýzu projektu. Je zde také registrován jeden posluchač (*IDebugEventSetListener*), který naslouchá a čeká na jakékoliv ukončení aplikace (*DebugEvent*.TERMINATE).

```
if (launch.getLaunchConfiguration().getAttribute(
    IJavaLaunchConfigurationConstants.ATTR_APPLET_PARAMETERS, "PathCoverage").
    equals("PathCoverage"))
      methodAnalyzer.schedule();
```

Výpis 2: Podmínka registrující námi spuštěný program

V této podmínce se kontroluje jestli se jedná o námi spuštěnou aplikaci parametrem, který se zadával při spouštění projektu jako JUnit ve třídě *JUnitRunner*. Na základě toho se pak spustí analýza daného projektu.

V původním modulu bylo toto spouštění analýzy vyřešeno trochu jinak a nastávaly zde chyby, které spouštěly analýzu vícekrát po sobě. Tímto řešením vytvořeným ve třídě *PathCoverageActivator*, se problémy spojené se spouštěním vyřešily.

Třída MethodAnalyzer: Celá analýza, která byla původně ve třídě *ButtonHandler* musela být rozdělená a přesunuta do této třídy. Samotná analýza je proces, který trvá nezanedbatelnou dobu a závisí na velikosti analyzovaného projektu. Z tohoto důvodu je lepší, aby takhle dlouho trvající operace probíhala na pozadí. K tomuto účelu je využitá třída *Job*. Naše třída *MethodAnalyzer* z ní pak dědí. Musí implementovat metodu *run*, do které musíme umístit potřebný kód. To nám zajistí vykonání tohoto kódu na pozadí. Do této metody tedy vložíme celý kód potřebný k analýze. Samotné spuštění analýzy se provede zavoláním metody *schedule()* na naší třídu. To probíhá ve třídě *PathCoverageActivator*, ve zmíněném posluchači, který čeká na ukončení běhu aplikace.

K tomuto účelu se sice v jazyce Java používá třída *ExecutionService*, ale třída *Job* umožňuje snadněji reagovat na ukončení celého procesu, pomocí registrace *JobChangeListeneru*.

Třída CoverageTools: Tato třída slouží ke komunikaci s druhým modulem. Obsahuje statické metody, které vrací referenci na potřebné třídy. Ty jsou vytvořené v třídě *PathCoverageActivator*. Přistupuje se k nim tedy přes instanci této třídy. Je tak vytvořený jednotný přístup k funkcím tohoto modulu.

5.3 Modul codecoverage.ui

Z důvodu rozdělení původního modulu na dva s cílem přesunout všechny prvky uživatelského rozhraní do jednoho modulu, byl vytvořen modul druhý. Byl pojmenován codecoverage.ui značící, že obsahuje prvky uživatelského rozhraní a komunikaci s ním. Jsou zde vytvořeny dva view, které zobrazují data o pokrytí. Dále pak spouštěcí tlačítko a třídy zodpovědné za anotace v editoru. V celém modulu pak byla použita novější technologie Eclipse 4 model.

5.3.1 Eclipse 4 model

Poslední zásadní změna pro Eclipse byla s uvolněním verze 3.0 při migraci k OSGi. Eclipse 4 model poskytuje značný odklon od Eclipse 3.x vydání, s uživatelským rozhraním reprezentovaným jako dynamický EMF model.

Eclipse 4 model(dále E4) poskytuje způsob, jak tvořit aplikaci jak ve fázi návrhu, tak za běhu. Díky oddělenému frameworku pro vykreslování, umožňuje reprezentaci uživatelského rozhraní pomocí různých nástrojů jako třeba JavaFX nebo HTML. Pro potřeby diplomové práce byl použit výchozí nástroj pro vykreslování SWT, který je úzce spojen s existujícím Eclipse 3.x uživatelským rozhraním. Aplikační model je pak tvořen souborem s příponou *.e4xmi. V tomto souboru se definují všechny prvky uživatelského rozhraní aplikace podobně, jako v souboru

plugin.xml. Základním prvkem tohoto souboru je *Model Fragment*. Ten pak tvoří základní kontejner pro vkládání dalších elementů uživatelského rozhraní. Dobrým zvykem je pak vytvořit si pro každou skupinu elementů se stejnými vlastnostmi(např. tlačítka, položky menu) oddělený *Model Fragment*. K úpravě aplikačního modelu, lze pak použít interaktivní formulář. Není tak zapotřebí přímo zasahovat do struktury *.e4xmi souboru, avšak i to je možné.

Druhou významnou změnou je, že již není nutné vytvářet podtřídy, které přispívají k infrastruktuře Eclipse. Místo toho se využívá tzv. „dependency injection"(podobně jak to děla Spring). Mohou se tak využívat služby platformy, odděleně od uživatelského rozhraní. Namísto odkazování na globální „singletony" prostřednictvím přístupových metod, je nyní používaná tzv. „injection" a tedy vkládání závislostí. To nám dovoluje vytvářet komponenty jako jednoduché POJOs (Plain Old Java Objects), tedy jako běžné objekty jazyka Java, které nejsou vázané žádnými zvláštními omezeními. To umožňuje jednodušší používání a také volnější vazbu na služby, které používají.

Díky těmto výše zmíněným výhodám, byla tato technologie použita ke tvorbě aktuálního modulu.

5.3.2 UIActivator

Tato třída plní stejný účel jako třída *PathCoverageActivator* v druhém modulu. Je tedy zodpovědná za celkový životní cyklus tohoto modulu. Obsahuje metody *start* a *stop*, ve kterých probíhá inicializace či odstranění proměnných či posluchačů.

V této třídě je registrován jeden zásadní posluchač. Ten má na starosti zobrazit potřebné *view*, obsahující informace o pokrytí. To musí být ale zobrazeno až po dokončení celkové analýzy. Bylo tedy třeba vytvořit posluchače na danou událost.

Celá analýza probíhá ve druhém modulu ve třídě *MethodAnalyzer*. Ta rozšiřuje třídu *Job* a implementuje tak její hlavní metodu *run*. Díky tomu dokážeme na tuto třídu registrovat požadovaného posluchače.

```
public void done(IJobChangeEvent event) {
        if (event.getResult().isOK()) {
           getWorkbench().getDisplay().asyncExec(new Runnable() {
              @Override
              public void run() {
                 showCoverageView();
              }
           });
        }
     }
```

Výpis 3: Posluchač registrující ukončení analýzy a zobrazující *CoverageView*

V posluchači *JobChangeAdapter* pak použijeme metodu *done*, která se vyvolá v okamžiku ukončení celého procesu analýzy. Po následné kontrole, jestli vše proběhlo v pořádku, se zavolá funkce, která nám zobrazí dané *view*.

Dále tato třída obsahuje metodu, která umožňuje „logování“ případných chyb. To je docíleno díky metodě *getLog()*, která je součásti třídy *AbstractUIPlugin*.

5.3.3 Spouštění analýzy

Ke spuštění jakékoliv akce z prostředí Eclipse je potřeba vytvořit tři následující prvky:

1. **Command** - reprezentuje námi definovaný příkaz, který identifikuje konkrétní akci.
2. **Handler** - je zastupován třídou, která obsahuje implementaci pro definovaný příkaz. Každý *handler* zastupuje, jeden příkaz. Více *handlerů* může být zastupováno jednou třídou.
3. **UI prvek** - sloužící ke spuštění daného příkazu z uživatelského prostředí Eclipse. Může to být položka v menu, tlačítko, či jiný prvek, ale také je možnost spustit příkaz programově. Každý prvek může být svázán pouze s jedním příkazem. Každý příkaz ale pak může být používán mnoha prvky.

Spolu tyto tři prvky tvoří v Eclipse mechanismus, kterým se spouští jakékoliv akce či události. Všechny tyto prvky jsou vytvořeny v souboru *fragment.e4xmi*.

RunJUnit command: Příkaz reprezentující akci spuštění projektu jako JUnit aplikace a následné analýzy. Nejprve bylo potřeba pro daný příkaz vytvořit v aplikačním modelu vlastní *fragment*. To bude později místo pro vytváření dalších příkazů. Pak už jen stačí vložit příkaz a definovat jeho jméno, případně popis. ID se pak vygeneruje automaticky podle jména.

RunJUnit handler: Obstarává vykonání samotné akce. Po vytvoření vlastního *fragmentu* pro všechny *handlery*, se vytvoří *handler* samotný. ID je znovu vygenerováno automaticky. Následně je třeba zadat příkaz, na který se má reagovat a třídu, která se postará o další zpracování. Třída pak musí obsahovat metodu, která bude provedena v případě vyvolání příkazu, který *handler* obsluhuje. Díky použití vlastnosti E4 není nutná implementace nějaké konkrétní metody s přesně definovaným názvem. Stačí vytvořit libovolnou metodu a přiřadit k ní anotací *@Execute*. To zajistí chování třídy jako *handler*.

Metoda pak spouští celý proces analýzy. Nejprve využitím třídy *ProjectParser* získá aktuální Java projekt. Pokud není tento projekt nalezen, zobrazí se varovné okno s hláškou. V opačném případě se nalezený projekt nastaví jako aktivní ve třídě *MethodAnalyzer* druhého modulu. Pak už se spustí celý proces, který zahrnuje spuštění serveru pro sběr dat, spuštění aplikace jako JUnit a následnou analýzu. Využívá se k tomu již zmíněná třída *CoverageTools*(viz. kapitola 5.2.3).

Spouštecí tlačítko: Prvek uživatelského rozhraní Eclipse spouštějící daný příkaz. I pro tuto skupinu elementů bylo třeba nejprve vytvořit vlastní *fragment*. Před samotným vložením tlačítka, které má být umístěné v přístrojové liště neboli *Toolbaru*, je potřeba vytvořit pár dalších elementů. Nejprve vytvoříme element *Trim Contribution*. Ten nám umožní vytvořit jako svého potomka *Toolbar*, kde už můžeme vytvořit element *Handled Tool Item*, reprezentující dané tlačítko. Zde můžeme nastavit mnoho vlastností, ale ty podstatné jsou: název, typ tlačítka(*push*), ikonu a hlavně příkaz ke spuštění, tedy *RunJUnit command*.

5.3.4 CoverageView

Pro zobrazení informací uživateli slouží v Eclipse *view*, v E4 nazývané jako *part*. *Parts* je obecný název pro *views*, *editors*, a další komponenty v E4.

CoverageView zobrazuje stromovou strukturu analyzovaného projektu. Jsou zde zobrazeny prvky *Project-Class-Method*. Dále je zde implementována funkčnost pro zobrazení dalšího *view* a komunikaci s ním. CoverageView je zobrazeno v prostředí Eclipse po ukončení analýzy spuštěného projektu. To je provedeno pomocí posluchače registrovaného ve třídě *UIActivator*(viz. kapitola 5.3.2).

Podobně jak tomu bylo u příkazu či tlačítka, je i pro *view* potřeba vytvořit vlastní *fragment* v aplikačním modelu. Do něho už pak lze umístit prvek *Part Descriptor* reprezentující potřebné *view*. Zde je znovu možné nastavit mnoho vlastnosti. Zde stačilo nastavit název, ikonu a URI k třídě obsahující implementaci pro tohle *view*. Každé *view* pak může obsahovat menu či nástrojovou lištu. CoverageView obsahuje nástrojovou lištu s jedním tlačítkem pro jeho aktualizaci. Tlačítko je vytvořeno stejným postupem jak je to popsáno v kapitole 5.3.3. Je tedy vytvořený *command*, *handler* s třídou obsahující implementaci a samotné tlačítko *Handled Tool Item*. Třída implementující samotnou aktualizaci využívá E4 servisu *IEventBroker* vloženou pomocí anotace *@Inject*. Ta poskytuje jednoduché API pro zasílání zpráv napříč platformou Eclipse. Zavoláním metody *post* na instanci této servisy se odešle asynchronní zpráva. Při komunikaci s UI komponentou je doporučeno používat zprávy asynchronní, aby nedošlo k zablokování UI vlákna. Je třeba pouze zadat parametr *topic* a tedy jednoduchý *String* identifikující zprávu a případně data typu *Object* jako druhý parametr. Odchytávání této zprávy pak probíhá v samotném *view*.

```
@Inject
@Optional
public void refreshViewer(@UIEventTopic("refresh") String data) {
   treeViewer.setInput(CoveragePlugin.getListOfProjects());
}
```

Výpis 4: Metoda pro aktualizaci *view*

Metoda je zde opět vložena pomocí anotace *@Inject*. Protože k zaslání zprávy vůbec nemusí dojít, je nutno použít anotaci *@Optional*. Parametr *@UIEventTopic* metody, pak dokáže odchytit

naší zprávu s příslušným identifikátorem *topic*. Aktualizace pak probíhá nastavením vstupních dat pro *treeViewer*, který nám stromovou strukturu zobrazuje. To nám způsobí překreslení této komponenty.

CoverageView obsahuje implementaci ve stejnojmenné třídě. Hlavní metoda této třídy *createPartControl* obsahuje vytvoření všech komponent a je u ní použitá anotace *@PostConstruct*. Tato anotace značí zavolání metody až po vytvoření samotné třídy, a vložení všech metod a parametrů, které byly do třídy vložené pomocí anotací. V této metodě se inicializuje komponenta *TreeViewer*, která zobrazuje stromovou strukturu projektů ve tvaru *Project-Class-Method*. Mimo různé vlastnosti týkající se formátování se komponentě musí nastavit *ContentProvider* a *LabelProvider*. Ty mají na starosti správné přiřazení názvu a obsahu k jednotlivým položkám této komponenty. K tomu byly vytvořeny třídy *ProjectContentProvider* a *ProjectLabelProvider* implementující potřebné rozhraní. Třídy pak obsahují potřebné metody, které přiřadí jednotlivým položkám název či ikonu(*ProjectLabelProvider*), či vrátí konkrétní objekt(*ProjectContentProvider*).

Pro spolupráci s dalším *view*, které zobrazuje podrobnější data o pokrytí dané metody, je potřeba komponentě *TreeViewer* registrovat dva posluchače:

1. **IDoubleClickListener** - naslouchá v zaregistrované komponentě a čeká na dvojklik. Po něm se v komponentě zobrazí (pokud již není otevřeno) druhé *view*.

2. **ISelectionChangedListener** - naslouchá v zaregistrované komponentě a reaguje na změnu výběru položek. Po změně vyšle událost, kterou pak odchytává druhé *view*.

První posluchač využívá metodu *doubleClick()*. V této metodě kontroluje jestli je druhé *view* již otevřené. Když tomu tak není, dané *view* zobrazí.

Druhý posluchač využívá metodu *selectionChanged()* s parametrem *SelectionChangedEvent*. Pokud dojde ke změně výběru, pomocí instance tohoto vstupního parametru se získá označená položka. Konkrétně zavoláním metody *getSelection()*. Protože lze označit více položek, musí se zavolat metoda *getFirstElement()*, k získání pouze jedné položky. Následně je potřeba pouze ošetřit, jestli námi označená položka je metoda, tedy instance třídy *Method*. K označení aktuálního výběru poslouží E4 služba *ESelectionService* opět vložená pomocí anotace *@Inject*. Nad instancí této služby pak stačí zavolat metodu *setSelection()* a jako parametr vložit získaný výběr položky. V druhém *view* pak existují metody, které tento výběr dokážou získat.

U všech posluchačů je nutné pamatovat na jednu zásadní věc. Pokud je registrován posluchač k nějaké komponentě, nesmí se taky zapomenout na jeho odstranění při zaniknutí této komponenty. To je provedeno v metodě *destroy()*, u které je použitá anotace *@PreDestroy*. Ta zajistí spuštění metody před samotným „zničením“ třídy. Běžně se tato metoda využívá při uvolňování zdrojů jako obrázky či různé soubory. Stačí tedy pouze nad komponentou *TreeViewer* zavolat metody pro odstranění těchto posluchačů.

5.3.5 PathCoverageView

PathCoverageView slouží k zobrazení detailních informaci o pokrytí dané metody a poskytuje ovládací prvky, pomocí kterých lze vykreslit jednotlivé cesty do editoru. Slouží tedy jako ovládací panel, poskytující dodatečné informace.

Jak bylo naznačeno v minulé kapitole, toto *view* spouští událost vyvolanou ve předchozím *view*. Pokud tak nenastane, vždy je možné otevřít kterékoli *view* pomocí menu *Window | Show-View*, kde si pak jde vybrat z požadované kategorie. Stejně jak tomu bylo u CoverageView, i PathCoverageView je třeba definovat v aplikačním modelu. Zde již do předem vytvořeného *fragmentu*(ten byl již vytvořen v minulé kapitole a obsahuje pouze *view*) vložíme prvek *Part Descriptor*. Znovu se nastaví ikona, název a URI ke třídě obsahující implementaci. Stejně tak i tohle *view* obsahuje vlastní nástrojovou lištu s jedním tlačítkem. To slouží pro odstranění všech anotací v editoru.

Stejnojmenná třída obsahující implementaci pro tohle *view* obsahuje taky metodu označenou anotací *@PostConstruct*, která má na starost vytvoření jednotlivých komponent. Toto *view* obsahuje více odlišných komponent a je tedy potřeba je nějakým způsobem uspořádat. To je docíleno nastavením „layoutu“ na *GridLayout*. Ten potřebuje ještě nastavit dva další parametry. Prvním je počet sloupců, který je nastaven na 2 a druhý definující stejnou šířku těchto sloupců(v našem případě *false*). PathCoverageView je tedy rozděleno na dvě části obsahující tyto komponenty:

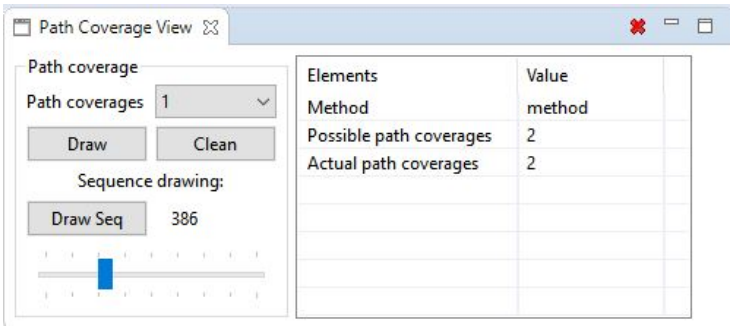
- **Ovládací část** - se skládá ze tří tlačítek a jednoho výběrového seznamu. Je zde taky posuvník, kterým lze nastavovat rychlost sekvenčního vykreslování cest.
- **Zobrazovací část** - tvoří tabulka s detailními informacemi o pokrytí metody.

PathCoverageView je vyobrazeno na Obr.10.

Tlačítka v ovládací části jsou tvořena z prvků *Button*. Pro definovaní akce po stisknutí tlačítka, je třeba ke každému registrovat posluchače na specifickou událost. Jde o třídu *SelectionListener* a jeho metodu *widgetSelected()*. Uvnitř této metody pak definujeme akci, kterou má dané tlačítko vyvolat. Pak je zde ještě výběrový seznam. Ten obsahuje jednotlivé pokryté cesty a je tvořen prvkem *ComboViewer*. Tomu se nastaví jednoduchý *ArrayContentProvider* umožňující zpracování pole. *LabelProvider* obsahuje pouze funkci, která vrací řetězec daného objektu. Při inicializaci *view* se jako vstup tohoto seznamu nastaví prázdné pole typu *Object*. Seznam se naplní konkrétními hodnotami až v okamžiku výběru nějaké metody. Posuvník zastupuje prvek *Scale* a slouží k regulaci rychlosti sekvenčního vykreslování. Tlačítka pak slouží k vykreslení, smazání či sekvenčnímu vykreslení konkrétní cesty vybrané v seznamu.

Zobrazovací část je pak tvořena tabulkou, kterou představuje komponenta *TableViewer*. Jak tomu je u všech zobrazovacích elementů, i zde je potřeba nastavit mnoho vlastnosti týkajících se vzhledu a zobrazení. K zobrazení daných informací v tabulce, byla vytvořena pomocná třída *MethodInfoData*. Ta obsahuje pouze pouze dvě vlastnosti *Title* a *Value*. Tato třída nám usnadní vkládání informací do tabulky. Díky tomu nastavíme tabulce *ArrayContentProvider*, který dokáže vložit do řádků tabulky položky z pole. Jednotlivé vlastnosti pomocné třídy jsou řetězce,

a proto nám metody v *ColumnLabelProvider* vrací přímo samotné objekty. Stejně jako výběrový seznam, je i tabulka při inicializaci *view* naplněná prázdným polem.



Obrázek 10: PathCoverageView

Pro odchytávání metody, která se vybere v CoverageView, se používá metoda *setSelection()*. Samozřejmě musí být označená anotacemi *@Inject* a *@Optional*, protože v okamžiku inicializace *view* ještě nemusí být vybraná žádná metoda. Vstupní parametr této metody je námi vybraná položka, která byla nastavená v předchozím *view* pomocí služby *ESelectionService*. Aby tento vstupní parametr ovšem obsahoval tuto selekci, je třeba před něho umístit anotaci *@Named(IServiceConstants.ACTIVE_SELECTION)*. To zapříčiní zavolání této metody po každém výběru. Uvnitř této metody pak jen ošetříme jestli zaslaný objekt není nulový a je instancí třídy *Method*. Pak se zavolá metoda *refreshMethodInfo()*, která naplní výběrový seznam a tabulku konkrétními daty a aktualizuje tak celé *view*.

5.3.6 AnnotationTools

Vykreslení jednotlivých pokrytých cest, je prováděno právě pomocí anotací, přímo do editoru. O tuto funkčnost se stará třída *AnnotationFactory*. Obsahuje metody, které vytváří či mažou anotace v editoru. Potřebuji k tomu instanci editoru a metody, a index cesty kterou mají vykreslit. Tyto metody nejprve získají instanci právě otevřeného dokumentu a hlavně jeho *IAnnotationModel*. Právě díky této instanci jsou jednotlivé anotace vykreslené do dokumentu. Informace o číslech řádku, které mají být anotovány a zbytek potřebných informací jsou obsaženy v třídě *Method* jejíž instance se do metody posílá. Samotné anotace jsou definovány v souboru *plugin.xml*.

Pro snadnější přístup k těmto metodám byla vytvořena pomocná třída. Dalším důvodem je jednotný přístup k UI vláknu. *AnnotationTools* obsahuje jednoduché metody, kterým se zadá jakou metodu a cestu je potřeba pokrýt a zbytek už je vyřešeno v této třídě. K získaní instancí editoru či dokumentu se použije jeden z nástrojů, který umožňuje přistoupit k UI vláknu. K tomu se využívá *UISynchronize* opět vložený do třídy pomocí *@Inject*. Pomocí instance tohoto nástroje lze zaslat asynchronní zprávu metodou *asyncExec()*. Jako parametr této metodě je nutno zadat novou instanci třídy *Runnable*, která obsahuje metodu *run()*. Zde se vloží kód, který potřebuje přistupovat k UI vláknu.

Poslední důležitou vlastnosti této třídy je její používání. Využívá vlastnosti E4 a lze ji jednoduše(stejně jak mnoho jiných E4 nástrojů a služeb) použít kdekoliv v aplikaci pomocí již známé anotace *@Inject*. Toho lze docílit dodržením těchto pravidel:

- Musí to být *non-abstract* třída, tedy neobsahující klíčové slovo *abstract*.
- Musí obsahovat *non-private* výchozí konstruktor.
- Třída musí být označena jako *@Creatable*.

To umožňuje vytvářet flexibilnější aplikace bez těsných vazeb mezí třídami či komponentami. O celý životní cyklus této třídy a propojení s jinými třídami se stará samotný E4 framework.

6 Zabalení a sestavení modulů

Nástroj CodeCoverage obsahuje dva moduly codecoverage.ui a codecoverage.core, které byly popsány v předešlých kapitolách. Tyto moduly poskytují veškerou funkcionalitu tohoto nástroje. Samostatně ovšem nedokážou fungovat správně. Pro jejich další distribuci je tedy potřeba z těchto modulů vytvořit jednotný celek.

Eclipse není jenom pouhá aplikace. Jeho architektura je založena na modulech(v Eclipse se označují jako plug-in), umožňující tak doinstalovat dodatečnou funkcionalitu. Moduly jsou pak seskupovány do větších celků, kterým se říká *features*. Jak moduly tak i *features* můžou být umístěny na tzv. *update site*. To umožňuje instalovat funkcionalitu do existující aplikace, nebo vytvořit aplikaci zcela novou. Tímto způsobem bude s těchto modulů vytvořen celistvý nástroj.

Tento nástroj je tvořen z modulů a komponent představených v minulém odstavci. Mimo jiné obsahuje několik dalších souborů, které umožňují bezproblémový chod tohoto nástroje v Eclipse. Jednou z posledních dílů skládačky k distribuci tohoto nástroje je automatické sestavení všech položek, které nástroj obsahuje. Většina modulů je v dnešní době sestavována pomocí nástroje Tycho[7]. Jedná se o Maven[8] *pluginy* určené pro sestavování Eclipse komponent.

6.1 Zabalení modulů

Samotné moduly poskytují veškerou funkcionalitu tohoto nástroje. K vytvoření jednotného nástroje, schopného další distribuce, je potřeba z těchto modulů vytvořit jeden celek. Táto část bude věnována tomu, jak toho lze docílit pomocí zabalení do *feature* a *update site*.

6.1.1 Vytvoření feature

Většina funkcí je v Eclipse poskytována skrze zásuvné moduly(*plug-ins*). Tyto moduly ovšem nejsou instalovány do Eclipse samostatně. V historii Eclipse platforma pracovala pouze s *features*, jejichž účelem je spojovat určitý počet modulů dohromady. V současné době lze instalovat moduly samostatně, ale veškerá funkcionalita používaná v Eclipse se instaluje přes *features*. *Features* se pak používají v konstrukci *update site*.

Feature Project se vytvoří přímo v prostředí Eclipse. Při vytváření je možnost, vybrat si ze seznamu moduly, které je potřeba spojit. Po dokončení se vytvoří projekt s jedním hlavním souborem, kterým je *feature.xml*. V tom lze následně upravit informace o jednotlivých modulech. Jsou zde i další elementy jako *description*, *license* či *copyright*. Tyto elementy nejsou povinné, ale pokud jsou vyplněny, zobrazují informace v průběhu instalace.

```
<feature
   id="codecoverage.feature"
   label="Path coverage tool for Eclipse"
   version="1.0.0.qualifier"
   plugin="codecoverage.ui">
```

```
    <plugin
    id="codecoverage.core"
    download-size="0"
    install-size="0"
    version="0.0.0"
    unpack="false"/>
</feature>
```

Výpis 5: Ukázka struktury souboru *feature.xml* s jedním modulem

Feature ID musí být globálně jedinečné, protože se používá jako identifikátor při instalaci v Eclipse. Verze komponent v Eclipse používají speciální formát *major.minor.micro.qualifier*. Tento formát používají moduly *features* a další komponenty Eclipse. Doporučuje se inkrementovat verze pomocí těchto pravidel:

- Inkrementace *major* verzí indikuje zpětně nekompatibilní změny
- Inkrementace *minor* verzí indikuje novou funkcionalitu se zpětnou kompatibilitou
- Inkrementace *micro* verzí indikuje opravu malých chyb

Qualifier může být jakákoliv textová hodnota reprezentující danou verzi. Pokud není specifikován Eclipse ho nahradí časovým razítkem, složeného z aktuálního data a času. Modul vypsaný výše (ve výpise kódu 5), má verzi 0.0.0. Když se pak *feature* publikuje, tato verze se nahradí aktuální verzí z modulu.

Pokud je potřeba *feature* rychle otestovat, bez nutností tvořit *update site*, lze přímo v prostředí Eclipse provést export. Po provedení tohoto kroku se vytvoří následující struktura souborů:

- artifacts.jar
- content.jar
- features/codecoverage.feature_1.0.0.201804140858.jar
- plugins/codecoverage.core_1.0.0.201804140858.jar

Při exportu se provede několik kroků. Nejprve se zkompilují všechny moduly, každý do vlastního JAR souboru. Následně se archivuje samotná *feature* a všechny položky se přesunou do odpovídajících složek *features* a *plugins*. V ukázce struktury je pouze *core* modul, ale ve skutečnosti se zde nachází i druhý modul *ui*. Soubory *artifact.jar* a *content.jar* obsahují XML soubory. Ty pak obsahují seznamy všech *features*, modulů a jejích závislosti, potřebných k instalaci.

Poslední věc, která byla vytvořena v naší *feature*, jsou tzv. *branding information*. Pokud se v Eclipse vyvolá dialog *About*(*Help* | *About*), zobrazí se okno s mnoha ikonkami tzv. top-level *features*, které byly nainstalované. Po kliknutí na jakoukoliv ikonu se zobrazí informace

o dané *feature*. Toho lze dosáhnout vytvořením souboru *about.ini* v jakémkoliv modulu, v tomto případě codecovarege.ui. Tento modul je pak potřeba nastavit v *feature* jako *branding plug-in*. Následně se už jen umístí potřebný text do daného souboru. Text v souboru se vypíše do atributu *aboutText* a ikona se specifikuje v atributu *featureImage*. Na závěr je potřeba se ujistit, že tento soubor, bude zabalen současně s modulem. To lze ověřit v souboru *build.properties*, který specifikuje složky a soubory k zabalení.

6.1.2 Vytvoření update site

Eclipse při instalaci umožňuje zobrazit jednotlivé *feature* či moduly v malých podmnožinách zvaných kategorie. Již samotné základní instalace Eclipse zahrnuje přes 400 *feature* a 600 modulů. Pokud by nějaký repozitář poskytoval podobný počet takových modulů, všechny by se pak při instalaci nového softwaru, zobrazovaly v jednom seznamu. Nevytvářelo by to zrovna přívětivé prostředí pro uživatelé. Z tohoto důvodu se jednotlivé moduly a *feature* zabalují do kategorií.

Pro přiřazení *feature* do kategorie, se vytvoří *Update Site Project*. V něm se pak vytvoří nová kategorie. Tu reprezentuje soubor *category.xml*(někdy taky *site.xml*). Dané kategorii se definuje ID, název a volitelně popis. Na závěr se přiřadí této kategorii codecovarege.feature. Tím se zajistí, že při instalaci nástroje, se zobrazí nejprve název kategorie *CodeCoverage* a po rozkliknutí, pak *feature* s názvem „Path coverage tool for Eclipse“.

6.2 Sestavení celého nástroje

Ještě pře samotným nahráním nástroje na vzdálený repozitář(bude popsáno v následující kapitole), je třeba provést sestavení. Celkový nástroj se totiž skládá se zmíněných *features*, *update site* a modulů , a navíc ještě konfiguračních souborů odpovědných za správný chod na dané platformě. Tyto soubory je potřeba použít, a vytvořit z nich fungující nástroj schopný další distribuce. K tomu je využíván nástroj Maven a jeho sada artefaktů Tycho.

6.2.1 Nástroj Maven + Tycho

Maven je často používaný nástroj pro sestavování aplikací vyvíjených nad platformou Java. Principem tohoto nástroje je vytvoření objektového modelu nad zdrojovým kódem, se kterým pak lze provádět různé operace. Nejčastěji jde o kompilaci, kontrolu, automatické spuštění testů či vytvoření balíků. Tento model je definován v souborech *pom.xml* (Project Object Model) a říká co a jak se má sestavit.

```
<groupId>codecoverage</groupId>
<artifactId>codecoverage.features</artifactId>
<version>1.0.0-SNAPSHOT</version>
<packaging>pom</packaging>
```

Výpis 6: Ukázka souboru *pom.xml* pro náš *feature* projekt

Tento soubor je ve formátu XML a obsahuje speciální hlavičku, která deklaruje že se jedná o POM. Nachází se v kořenovém adresáři každého projektu. Zde se pak spouští příkaz *mvn* nástroje Maven, který načte informace s tohoto souboru a provede uživatelem definovaný cíl (definuje se za příkazem *mvn* např. *mvn clean package* pro vyčištění a následne zabalení projektu). Maven projekty mají `groupId`, `artifactId` a `version`, které pomáhají k identifikaci. Pak je zde ještě atribut `packaging`, který definuje typ zabalení. Výchozi typ pro sestavování java projektů je JAR. Při použití Tycho, budeme potřebovat mnohem více typů zabalení.

Tycho je sada balíčků, sada Maven *pluginů* a rozšíření, umožňující sestavování Eclipse komponent. Tycho podporuje sestavení Eclipse modulů, OSGi balíků, Eclipse *features* a *update site*. Dokáže maximálně využívat metadata jednotlivých Eclipse komponent. Pro získání všech závislostí modulu využívá například *MANIFEST.MF* soubor, který tyto závislosti obsahuje.

6.2.2 Vytvoření struktury projektů

Nástroj CodeCoverage je tvořen několika projekty. Je dobrým zvykem, nejenom v prostředí Eclipse, jednotlivé soubory či projekty nějak logicky strukturovat, podle jejich účelu či společných vlastnosti. Vytvořena struktura se následně využije při vytváření jednotlivých souborů POM, které nástroj Maven potřebuje k sestavení celého nástroje. Všechny soubory a projekty byly umístěny do následujících složek podle jejich významu:

- **bundles** - slouží pro všechny moduly. Obsahuje *codecovarege.core* a *codecoverage.ui*.
- **features** - slouží pro všechny *features*. Obsahuje *codecoverage.feature*.
- **releng** - tento soubor(z angl. release engineering) se skládá z následujících položek.
  - konfigurační projekt *codecoverage.configuration*. Obsahuje informace potřebné k sestavení.
  - projekt *codecoverage.target*. Obsahuje informace o cílové platformě.
  - *update site* projekt *codecoverage.update*. Definuje kategorii a dále obsahuje skripty pro odeslání na vzdálený repozitář.

Celá tato struktura je obsažena v jednoduchém Eclipse projektu, který tvoří tzv. rodičovský projekt. Nese jednoduše označení *codecoverage*.

6.2.3 Typy POM souborů

Jak už bylo vysvětleno, sestavování pomocí Tycho se nastavuje pomocí standardního Maven konfiguračního souboru *pom.xml*, často nazývaného POM soubor. Tento konfigurační soubor je potřeba umístit ke každému projektu, a navíc taky do každé složky. Soubory se pak musí vzájemně prolínat a odkazovat na sebe. Tímto způsobem se vytvoří z těchto souborů jeden celek. Pro sestavování pomocí Tycho je typické použít dva speciální POM soubory. Prvním je

konfigurační POM, obsahující hlavní nastavení. Druhý pak zahrnuje všechny Eclipse komponenty (*modules*), které jsou relevantní pro sestavení.

Tycho také umožňuje vygenerovat některé POM soubory, založené na informacích získaných z metadat jednotlivých komponent. Říká se mu Tycho POM-less build. Všechny soubory ovšem vygenerovat nedokáže a proto jich se musí vytvořit ručně.

Definice souboru pro POM-less Tato technologie umožňuje vynechání některých POM souborů. Tyto soubory ovšem negeneruje, nýbrž pouze čerpá informace z metadat jednotlivých komponent. K tomu potřebujeme vytvořit speciální soubor *extensions.xml*, který se umísti na úroveň rodičovského projektu do složky *.mvn*. Obsah souborů vypadá takto (viz výpis kódu 7).

```
<extensions>
  <extension>
    <groupId>org.eclipse.tycho.extras</groupId>
    <artifactId>tycho-pomless</artifactId>
    <version>1.0.0</version>
  </extension>
</extensions>
```

Výpis 7: POM-less definiční soubor *extensions.xml*

Dále pak existují pravidla, kterými se jednotlivé informace mapují:

- `packaging` - *eclipse-plugin* pokud se najde soubor *MANIFEST.MF*, nebo *eclipse-feature* pokud se najde *feature.xml*.
- `groupId` - stejné jako rodičovský POM
- `artifactId` - pro modul se použije *Bundle-SymbolicName* z *MANIFEST.MF*. Pro *feature* pak ID ze souborů *feature.xml*.
- `version` - použije se *BundleVersion* z *MANIFEST.MF* nebo verze ve *feature* z *feature.xml*.

K sestavení je vyžadováno aby verze, které se nacházejí v souborech POM a jednotlivých Eclipse komponentách, byly stejné. Důležité je také zmínit že *.qualifier*, který se používá v Eclipse komponentách se automaticky mapuje na *-SNAPSHOT*, který se používá při Maven sestavování. K synchronizaci těchto verzí slouží následující příkaz:

mvn -Dtycho.mode=maven org.eclipse.tycho:tycho-versions-plugin:update-pom

Typy jednotlivých komponent Jak již bylo zmíněno, Maven používá ve výchozím stavu k sestavení typ JAR. Tento typ se určuje v atributu `packaging`. Tycho pak definuje několik druhů těchto atributů, závisle na tom, o jakou Eclipse komponentu jde. Pro náš účel byly použity následující druhy atributů:

- *eclipse-plugin* - bylo použité pro moduly
- *eclipse-feature* - bylo použité pro *feature*
- *eclipse-repository* - bylo použité pro *update site*
- *eclipse-target-definition* - bylo použité pro POM soubor definující cílovou platformu

Mimo tyto 4 druhy je ještě využíván druh *pom*. Ten se využívá u konfiguračního souboru nebo jednotlivých rodičovských POM souborech.

6.2.4 Vytvoření POM souborů

Jak bylo zmíněno, existuje tedy několik typů POM souborů. Rozlišují se hlavně podle atributu `packaging`, případně podle toho kde jsou umístěny, či jakou funkci zastávají.

V každém POM souboru lze dále definovat různé konfigurační vlastnosti související s daným projektem. Lze také použít různé jiné funkce poskytované některým z balíčků Tycho či přímo Maven artefakty. Aby nebylo potřeba definovat tyto konfigurační vlastnosti v každém souboru zvlášť, je dobré vytvořit z těchto souborů strukturu, a všechny konfigurační položky přenést do hlavního souboru. Jednotlivé POM soubory se pak mezi sebou odkazují pomocí atributu `parent`. Dovnitř tohoto atributu se pak specifikují `groupId`, `artifactId` a `version` POM souborů z kterého ten aktuální vychází. Tento princip je podobný dědičnosti v objektově orientovaném programování. Pokud se rodičovský POM nenachází o úroveň výš od aktuálního POM souboru, musí se navíc definovat relativní cesta k tomuto souboru pomoci atributu `relativePath`.

Hlavní POM soubor Tento soubor se nachází kořenovém adresáři všech projektů, tedy v rodičovském projektu *codecoverage*. Jeho název je *codecoverage.root*, definovány v atributu `artifactId`. `groupId` mají všechny soubory stejné a to *codecoverage*. Pomocí atributu `module`, definuje části ze kterých se tento nástroj skládá. Obsahuje tedy tři položky *bundles*, *features* a *releng*. Tento soubor je sice hlavní POM, ale má ještě definovaného rodiče. Jde o konfigurační soubor. Ten spolu z hlavním tvoří dva důležité soubory celého sestavení.

POM-less rodič pro moduly Využití technologie POM-less nám zmenší počet souborů, které musíme vytvořit. Pro jednotlivé moduly tedy soubory tvořit nemusíme a o všechno se postará balíček definovány v *.mvn/extensions.xml*. Jelikož máme projekty strukturované do složek, potřebujeme pro každou složku vytvořit další soubor. Ten propojí tuto složku a obsahující projekty z hlavním POM souborem. Tento soubor se označuje jako POM-less rodič a nese název *codecoverage.bundles*. Kromě propojení na hlavní POM ještě obsahuje oba moduly *codecoverage.core* a *codecoverage.ui* v atributu `module`. Atribut `packaging` tohoto souboru je, stejně jako u hlavního, nastaven na *pom*. U jednotlivých modulů pak POM-less použije *eclipse-plugin*.

POM-less rodič pro features Stejně jak tomu bylo u modulů, i zde je třeba vytvořit POM-less rodiče, který propojí tuto složku s hlavním POM. Jeho název, stejně jak u ostatních POM, je definovány v atributu `artifactId` - *codecoverage.feature*. Obsahuje stejnojmennou *feature* zapsánou v atributu `module`. Typ souboru v atributu `packaging` je znovu *pom*. U *feature* projektu by pak měl být *eclipse-feature*.

POM pro releng Tato složka již neobsahuje POM-less rodiče. Jednotlivé položky v této složce totiž potřebují ručně definovaný POM soubor. Tento soubor ovšem obsahuje stejné atributy jako jeho předchůdci, v předchozích dvou složkách. Název znovu odpovídá názvu složky. Fyzicky táto složka obsahuje tři projekty. V atributu `modules` jsou ale definovány jenom dva z nich - *codecoverage.target* a *codecoverage.update*. Třetí je konfigurační projekt z již zmíněným konfiguračním POM souborem. Na tento soubor navazuje hlavní POM a je tedy pomyslným „kořenem“ celé struktury těchto souborů.

POM pro target definition Tento projekt slouží pro definici cílové platformy. Jeho POM soubor obsahuje název *codecovcerage.target.neon*, značící cílovou platformu. Atribut `packaging` pak obsahuje typ specifický pro projekty z tímto účelem - *eclipse-target-definition*. Jako rodič je zde uveden POM soubor vytvořený ve složce *releng*.

POM pro update site Tento projekt již byl detailněji popsaný na začátku aktuální kapitoly. Jeho POM soubor obsahuje název *codecoverage.update* a odkazuje na stejného rodiče jako POM pro definici cílové platformy. Jako `packaging` je zde uveden *eclipse-repository*. Dále obsahuje další položky, které souvisí s nahráním celého nástroje na vzdálený repozitář. Detailní popis tohoto POM souboru bude v následující kapitole.

Konfigurační POM Tento projekt je konfigurační a obsahuje pouze POM soubor. V tom se nachází důležité informace potřebné ke kompletnímu sestavení. Název tohoto POM je *codecoverage.configuration* a atribut `packaging` je pak *pom*. Detailnější popis tohoto souboru je v následující sekci.

6.2.5 Konfigurační projekt

Z důvodu umístění všech důležitých nastavení, bylo potřeba vytvořit konfigurační projekt. Jde o čistý Eclipse projekt, který obsahuje pouze definiční soubor *.project* a samotný POM soubor obsahující veškerou konfiguraci. V tomto souboru je pár důležitých atributů, které je potřeba zmínit.

Hned na začátku souboru je atribut `properties`. Ten obsahuje definované proměnné, které pak lze použít, díky zmíněném propojení, v každém námi vytvořeném POM souboru. Uvádí se zde například verze některých důležitých artefaktů, nebo třeba kódování. Vytvoření této proměnné je jednoduché. Stačí pouze uvést název a dovnitř atributu pak hodnotu. Přístup k ním

je pomocí následující notace: ${nazev.promenne}.

Dále následuje atribut `build`. Zde jsou pak definovány všechny artefakty(v atributu `plugin`), které jsou potřebné k sestavení. Mimo to je zde sekce `pluginManagement`. Tato sekce umožňuje sdílet nastavení jednotlivých artefaktů mezi zbylými POM soubory. Mezi důležité artefakty patří například *tycho-packaging-plugin*. S jeho pomocí definujeme z čeho má být složený název výsledných komponent a celkové zabalení. Dále pak *maven-jarsigner-plugin*, který se stará o podpis tohoto nástroje. Pomocí *target-platform-configuration* artefaktu, se specifikuje projekt, zodpovědný za konfiguraci cílové platformy. Součástí nastavení tohoto artefaktu je rovněž konfigurace prostředí. Atribut `environment` specifikuje operační systémy, na kterých je možné nástroj nainstalovat.

6.2.6 Definice cílové platformy

Každý nástroj či modul vyvinutý pro prostředí Eclipse, potřebuje ke svému chodu využít další moduly, či nástroje. Pro využití komponent uživatelského rozhraní se například využívá modul *org.eclipse.ui*. Závislosti na použité moduly jsou v projektu uvedeny v souboru MANIFEST.MF. Zde jsou uvedeny názvy jednotlivých modulů a jejich verze. Je pak nutné použít správnou verzi modulu tak, aby byl podporován v použité verzi Eclipse.

Ve výchozím stavu je v prostředí Eclipse nastavená cílová platforma, obsahující všechny moduly, které byly kdy v platformě použity. Z tohoto důvodu bylo potřeba vytvořit si definici cílové platformy vlastní. Ta pak obsahuje balíky modulů (*features*), které jsou potřebné pro správný běh nástroje CodeCoverage. Tato definice je v separátním projektu, který obsahuje pouze soubor s danou definicí a dříve zmíněný *pom.xml*6.2.4 se speciální konfiguraci.

Nástroj byl vyvíjen pouze na platformě Eclipse 4.6 (Neon), obsahuje proto definici pouze pro tuto cílovou platformu. To je docíleno přidáním již zmíněných *features* s moduly, které byly použity. Pro zajištění správných verzí pro konkrétní platformu, je nutno použít repositář specifický pro danou verzi Eclipse [6].

6.2.7 Podpis nástroje

Při instalaci nástroje, Eclipse upozorní jestli je daný nástroj „podepsaný“, či ne. Digitální podpis zajišťuje, že se obsah daného nástroje nezměnil. Podpisem je také verifikována identita osoby či autority.

K podpisu obsahu je nutné použít privátní a veřejný klíč. Privátní klíč se použije na podpis obsahu. K verifikaci, že nedošlo k modifikaci obsahu, pak klíč veřejný. K tomuto účelu se využívá Java *keytool* nástroj. Pomocí parametru*genkey* se vygeneruje pár klíčů. Nástroj se spouští z příkazové řádky. K správnému vygenerování je potřeba specifikovat následující parametry:

- *alias* - označení pro klíč.
- *keypass* - heslo pro privátní klíč.

- *keystore* - cesta k souboru s klíči. Soubor *cacert*, který obsahuje všechny klíče, se nachází v místě instalace JDK v podsložce security.
- *storepass* - heslo k přístupu do souboru *cacert*.
- *dname* - rozlišovací jméno(z angl. distinguished name) identifikující vlastníka. Je reprezentováno několika páry jméno=hodnota oddělenými čárkou. Definují se zde např. jméno (*cn*), „organizační jednotka“ (*ou*), organizace (*o*), lokalita (*l*), či stát (*s*).

Po vygenerování klíče je potřeba provést samotný podpis obsahu. Ten je možné provést také z příkazové řádky. Pro naše účely byl ale použit již zmíněný balíček *maven-jarsigner-plugin*. Je specifikován v konfiguračním POM souboru. V sekci `executions` má specifikovaný cíl *sign*, který tento podpis provede.

Pro správnou funkci tohoto balíčku je potřeba specifikovat všechny parametry, nutné k podpisu. Mezi parametry jsou i hesla ke klíčům. Bylo potřeba tyto parametry specifikovat na místě, kde nebudou viditelné pro ostatní. K tomuto účelu slouží v nástroji Maven soubor *settings.xml*, který se nachází v instalaci samotného nástroje. Zde se pak specifikují různé položky, které chce uživatel skrýt před ostatními(např. přístupová hesla nebo tajné klíče k přístupu na vzdálený repozitář). Vytvoří se zde profil s potřebnými parametry. Při samotném kompilování či uvolňování nástroje na vzdálený repozitář se pouze spustí parametr aktivující tento profil. To zajistí modulu přístup k potřebným konfiguračním položkám. Detailní popis uvolňování nástroje na vzdálený repozitář, bude popsán v následující kapitole.

7 Vytvoření skriptu pro uvolnění nástroje na P2 repozitář

Poslední fázi vývoje nástroje CodeCoverage je uvolnění na vzdálený repozitář. Existuje mnoho cest jak lze tuto akci provést, v závislosti na použitém repozitáři a službě, pomocí které bude repozitář dostupný.

Eclipse platforma poskytuje mechanismus pro instalaci a aktualizaci funkcionality nazývaný *Eclipse P2*. Pro tento účel musí být nástroj správně zabalený, aby vytvářel tzv. P2 repozitář (někdy označován jako P2 update site). Nástroje CodeCoverage je distribuován v tzv. P2 kompozitním repositáři, který poskytuje širší funkcionalitu. Problematika vytvoření tohoto nástroje je popsána v první části této kapitoly.

Samotnou instalaci nástroje do prostředí Eclipse je možné provést lokálně. K poskytnutí nástroje širší komunitě uživatelů Eclipse je nutné umístění na nějaký webový server. Pro distribuci daného repozitáře byla vybrána služba JFrog Bintray[9]. Jedná se o službu umožňující šířit Open Source aplikace. V další části kapitoly budou stručně popsány možnosti, jak lze hotový nástroj na tuto službu umístit.

Maven a Tycho umožňují automatické uvolňování všech artefaktů na specifikovaný repozitář. Obvykle prováděné pomocí příkazu *mvn deploy*. Tímto způsobem se ale nasazují pouze Maven artefakty. Pro uvolnění nástroje CodeCoverage byly použity jiné technologie. Konkrétně nástroj cURL[11] a REST API služby JFrog Bintray. Spolu se Ant skripty[13] se starají o vytvoření struktury celého kompozitního repozitáře a jeho uvolnění. Celý tento proces je pak navázán na Maven/Tycho „build“. Vytvoření těchto skriptů a provázání bude popsáno v závěru této kapitoly.

7.1 Eclipse p2 kompozitní repozitář

Originální mechanismus umožňující aktualizace v prostředí Eclipse byl ve verzi 4.3 nahrazen novým, nazývaným P2(Provisioning Platform)[10]. Poskytuje výkonnější mechanismy a zahrnuje možnost aktualizovat nativní kód a konfigurační soubory. Úložiště, ze kterého je prováděná instalace či aktualizace se pak označuje jako P2 repozitář.

Obecně je repozitář tvořený z instalovatelných jednotek, které jsou sestavený z modulů a *features* případně organizovaných do kategorie. V Eclipse je takový repozitář tvořen *update site* projektem(viz. kapitola 6.2.1). Tyto názvy jsou proto někdy zaměňovány. Jak již bylo popsáno v kapitole věnované *update site*, po vytvoření a sestavení tohoto projektu se kromě složek *features* a *plugins* s jednotlivými položkami, vytvoří také soubory *artifact.jar* a *content.jar*. Tyto soubory obsahují P2 data(hlavně závislosti na moduly používané nástrojem CodeCoverage), které jsou potřebné k instalaci.

P2 umožňuje vytvořit kompozitní repozitář, který dokáže agregovat více *update site* projektů dohromady. Tato možnost se hodí v případě tvorby větších projektů, sestavených z více *update site*. Kompozitní repozitář může být také použit k vytvoření konzistentní „top-level“ stránky, agregující různé verze jednoho nástroje. Tato technika umožňuje ukládat různé verze do oddě-

lené složky. Pro přístup ke konkrétní skupině verzí, stačí správně specifikovat adresu a jednotlivé vydání se prezentují jako oddělené úložiště. Například pomocí URL *cesta/k/repository/1.0*, vybereme pouze verze produktů 1.0.x. Cílem pro použití kompozitního repozitáře bylo vytvořit jednotlivé verze nástroje, odladěné pro konkrétní vydání Eclipse. Všechny verze by tak byly dostupné pod jedním URL, s malými modifikacemi.

Struktura kompozitního repozitáře obsahuje dva hlavní soubory *compositeArtifacts.xml* a *compositeContent.xml*. Oba pak mohou být zabaleny ve stejnojmenných souborech s koncovkou *.jar*. Tyto soubory jsou praktický identické a liší se pouze v typu repozitáře deklarovaném v sekci `repository`. V sekci `children` a obsahujících atributech `child` se pak specifikují jednotlivé verze. Kompozitní repozitáře je možné řetězit a tedy odkazovat na další kompozitní repozitář. Dále zde může být soubor *p2.index*, který obsahuje záznam o přítomnosti dvou výše zmíněných souborů. Mechanismus P2 při instalaci hledá jako první právě soubor *p2.index*. Struktura repozitáře použita pro nástroj CodeCoverage je naznačena na Obr. 11 Složka *releases* obsahuje

```
root
|-- releases
|   |-- 1.0.0.v2018...
|   |   |-- artifacts.jar
|   |   |-- content.jar
|   |   |-- features
|   |   |   |-- codecoverage.feature_1.0.0.201803151354.jar
|   |   |-- plugins
|   |   |   |-- codecoverage.core_1.0.0.201803151815.jar
|   |   |   |-- codecoverage.ui_1.0.0.201803151815.jar
|   |-- 1.1.0.v2018...
|   |-- 1.1.1.v2018...
|   |-- 2.0.0.v2018...
|   ...
|-- updates
|   |-- compositeContent.xml
|   |-- compositeArtifacts.xml
|   |-- 1.0
|   |   |-- compositeContent.xml
|   |   |-- compositeArtifact.xml
|   |-- 1.1
|   |   |-- compositeContent.xml
|   |   |-- compositeArtifact.xml
|   |-- 2.0 ...
|   ...
|-- zipped
    |-- codecoverage.update-1.0.0.201804171157.zip
    |-- codecoverage.update-1.0.0.201804130850.zip
```

Obrázek 11: Struktura repozitáře nástroje CodeCoverage

jednotlivé P2 repozitáře, každý ve vlastní složce pojmenované podle jejich verze. Druhá složka *updates* obsahuje tzv. *main composite*. Ten pak odkazuje na složky označené podle *major.minor* verzí. Ty jsou nazývané jako *child composite* a odkazují na jednotlivé verze jednoduchých P2 repozitářů. Složka *zipped* pak obsahuje jednotlivé verze nástroje zabalené do archívu.

7.2 Způsob nahrávání na webový server

Pro distribuci nástroje CodeCoverage je používaná služba JFrog Bintray [9]. Tato služba umožňuje hlavně zmíněnou distribuci požadovaného nástroje, podporující hlavní formáty balíčků jako jsou Docker, Vagrant, Maven, či NuGet. Mimo to umožňuje uložit jakoukoliv strukturu souborů, které jsou dostupné pod daným odkazem. Poskytuje tak možnost vytvoření kompozitního repozitáře.

Služba umožňuje několik způsobů pro nahrání potřebných souborů. Nejjednodušší způsob je použít webové rozhraní a soubory nahrát ručně. Tento způsob je poměrně zdlouhavý a pracný. Další možnosti je použití REST API této služby. Tato služba je založena na protokolu HTTP. Díky využití tohoto protokolu a jedné z jeho operací (*PUT*), je pak provedeno nahrání konkrétních souborů.

K zaslání samotného HTTP požadavku je potřeba využít nástroje cURL[11]. Tento nástroj slouží k přenosu dat a podporuje mnoho protokolů, mezi jinými i zmíněný HTTP. Lze ho spustit z příkazové řádky a použitím různých parametrů pak poskládat potřebný HTTP požadavek. Celkový proces nahrání souboru na Bintray zahrnuje použití REST API a nástroje cURL. Výsledný příkaz pak umožní nahrání jednotlivých souborů, specifikováním potřebné cesty, s cílem vytvoření struktury kompozitního repozitáře [13].

7.2.1 Využití Bintray REST API

Bintray umožňuje nahrát obsah nástroje použitím REST API. Nahrávání je jenom jednou z mnoha možností tohoto API, avšak jinou funkcionalitu není potřeba využívat. Před samotným nahráním na server je potřeba pomocí uživatelského rozhraní vytvořit repozitář z typem *generic* a v něm pak balíček pojmenovaný *releases*. Do tohoto balíčku pak bude nahrávána celá struktura souborů kompozitního repozitáře. Následující příkaz využívá REST API služby, s cílem nahrání specifikovaného souboru.

```
PUT /content/:subject/:repo/:file_path;bt_package=:package;bt_version=:version;publish=1
```

Jednotlivými parametry pak specifikujeme co chceme nahrát a kam to chceme nahrát. Parametr *content* specifikuje že se jedná o nahrávání obsahu. Samotný obsah se pak specifikuje až pomocí cURL. Nejprve se specifikuje *subject*, tedy login uživatele a jméno repozitáře *repo*. Dále je třeba určit cestu, kde se má soubor uložit - *file_path*. Tato cesta určuje rozmístění do jednotlivých složek struktury kompozitního repozitáře. Nakonec se specifikuje *package* a verze *version* aktuálního sestavení. *Package* se uvádí pokaždé *releases*, protože se soubory nahrávají stále do stejného místa. Na úplný závěr příkazu se specifikuje akce *publish=1*, pro automatické publikování souboru ihned po uložení na server.

7.2.2 Vytvoření příkazu pro nahrávání pomocí cURL

Pro odeslání obsahu na vzdáleny server služby Bintray lze použít zmíněné REST API. Toto API pracuje s HTTP operací PUT. K odeslání tohoto požadavku je nutné využít nástroj cURL. Ten slouží k přenosu dat a podporuje mnoho známých protokolů, mezi něž patří i HTTP. Pomocí několika speciálních parametru se následně vytvoří výsledný příkaz.

```
curl -XPUT -T $f -u{BINTRAY_USER}:{BINTRAY_API_KEY} "https://api.bintray.com/
    content/{BINTRAY_OWNER}/{BINTRAY_REPO}/{TARGET_PATH}/$f;bt_package={
    PCK_NAME};bt_version={PCK_VERSION};publish=1"
```

Výpis 8: Příkaz pro odeslání obsahu na vzdálený server

Tento příkaz tedy využívá nástroj cURL, společně s operací HTTP protokolu PUT a speciální příkazem specifikovaným v předchozí sekci. Na začátku příkazu se specifikuje metoda HTTP požadavku, pomocí parametru -X. Dále pomocí parametru -T soubor potřebný k odeslání. V ukázce 8 pouze symbolicky jako $f. Poslední parametr -u přiloží informaci o uživateli spolu s speciálním API klíčem. Nakonec je již zmíněné URL, specifikující cestu k umístění souboru. V další podkapitole bude popsán skript, který využívá tohoto příkazu k automatickému odeslání všech potřebných souborů.

7.3 Skript pro uvolnění nástroje

Po sestavení, se nástroj CodeCoverage skládá z několika souborů. Ty se následně nahrají na webový server pomocí výše popsaného příkazu. K automatizaci tohoto kroku se používají Ant skripty[13]. Celá operace je pak provázána a spouštěna při Maven/Tycho sestavování.

Celý tento proces se skládá ze tří základních části. Tyto části jsou přiřazeny k jednotlivým fázím sestavování. Všechny části se spouští z POM souboru *update site* projektu. Po provedení tohoto procesu, jsou všechny potřebné soubory odeslány na server.

7.3.1 Fáze prepare-package

K této fázi je přiřazen první krok celého procesu. Ten má za úkol stáhnout ze vzdáleného serveru, určeného k nahrávání celého nástroje, kompozitní metadata pro *main* a *child* kompozit. Pokud tyto metadata nenalezne, znamená to, že kompozit ještě nebyl vytvořen. V případě absence *child* kompozitu, který obsahuje jednotlivé *major.minor* verze, znamená publikaci verze nové. Pokud jsou metadata nalezena, stáhnou se do odpovídajících složek do adresáře *target*. Zde pak budou použity k dalšímu zpracování.

Celý tento krok je spouštěn pomocí *maven-antrun-plugin* ve fázi *prepare-package*. Tento artefakt pak dokáže spustit specifický Ant skript, který se nachází v souboru *bintray.ant*. To je provedeno zavoláním cíle *get-composite-metadata*.

Jakékoliv úkoly prováděné v Ant skriptech jsou umístěny v kontejneru `target`. Úkoly můžou být obsaženy i v jiných elementech(třeba makrech), ale spuštění se provádí pomocí `target`.

Spuštěním tohoto cíle pro stažení metadat obou kompozitů se spustí dva makra a jeden další cíl. Makra jsou definovány v sekci `macrodef`. Obsahují atributy, které definují cestu k jednotlivým souborům. První makro *get-metadata* pouze zavolá druhé makro *get-file*, které stáhne specifikované soubory. Nakonec se zavolá cíl, který ve stažených metadatech změní hodnotu jedné proměnné na *false*, pro umožnění případného přidání dalšího potomka do kompozitu.

Všechny potřebné adresy a přístupové informace k serveru jsou definovány v proměnných. ty se nachází jak v POM souboru *update site* projektu tak v jednotlivých Ant souborech. Tajné informace k autentifikaci na straně serveru jsou pak uloženy v souboru *settings.xml*.

7.3.2 Fáze package

Druhý krok celého procesu nahrávání je přiřazen k fázi *package*. Jeho cílem je vytvořit nebo aktualizovat kompozitní repozitář stažený v prvním kroku. Pokud se repozitář nestáhne, tudíž nebyl ještě vytvořen a musí ze vytvořit kompozitní repozitář nový. V opačném případě se jednotlivé soubory aktualizují a přidají se zde záznamy právě sestavované verze.

K vytváření a aktualizaci kompozitního repozitáře slouží znovu Ant skript. Tentokrát je ale spuštěn pomocí artefaktu *tycho-eclipserun-plugin*. Ten slouží ke spuštění Eclipse procesu s libovolnými parametry příkazové řádky. Tento artefakt nám ke svému spuštění umožní použít další artefakty definované pomocí `dependencies`. Ty pak poskytují funkce, které jsou použity k vytvoření kompozitního repozitáře. Samotné spuštění tohoto kroku, se provede zavoláním Maven cíle (`goal`) - *eclipse-run*. Díky elementu `appArgLine` se spustí potřebný Ant soubor s definovaným cílem (`target`) a potřebnými parametry(*-Dname=value*).

```
<appArgLine>-application org.eclipse.ant.core.antRunner -buildfile packaging-
    p2composite.ant p2.composite.add -Dname=value</appArgLine>
```

Výpis 9: Parametry pro spuštění Ant souboru zadané v `appArgLine`

Spuštěný cíl *p2.composite.add* obsahuje volání makra pro *main* a *child* kompozit. Toto makro pak s využitím „Ant tasku“ *p2.composite.repository* importovaného pomocí závislosti v souboru POM, dokáže vytvořit či aktualizovat metadata repozitáře. Na konci tohoto makra je vytvořen soubor *p2.index*.

7.3.3 Fáze verify

Poslední krok celého procesu je přiřazen k fázi *verify*. Cílem tohoto kroku je nahrát všechny soubory na vzdálený server. Budou tak nahrány soubory vytvořené při sestavování nástroje a taky metadata vytvořené či aktualizované v předchozích krocích. Na konci tak bude vytvořená struktura souborů demonstrována na Obr. 11.

Ke spuštění tohoto kroku je stejně jako u prvního použit *maven-antrun-plugin*. Tentokrát zavoláním cíle *push-to-bintray*, umístěného v souboru *bintray.ant*.

Tento cíl následně volá další cíle. Prvním z nich je *postprocces-metadata*. Ten pouze nastaví

proměnnou, která umožňovala přidání jednotlivých potomků do kompozitu v prvním kroku, zpátky na *true*. V další části jsou již volány cíle, které odesílají obsah na server. Odeslání souborů provedeme pomocí příkazu 8, popsáného v minulé podkapitole. To je provedeno pomocí elementu `apply`, který dokáže spustit systémový příkaz a speciálních „mapperů“, díky nimž složíme výsledný příkaz dohromady. Všechny parametry potřebné pro odeslání příkazu jsou definovány v aktuálním Ant souboru v elementech `property`.

8 Celý proces sestavení a uvolňování nástroje CodeCoverage

Pro sestavení celého nástroje CodeCoverage byl využíván Maven/Tycho. Tycho je sada balíčků, sada Maven artefaktů a rozšíření, umožnující sestavení Eclipse komponent(viz. kapitola 6.2.1). Celý proces sestavování a uvolňování je spuštěn pomocí nástroje Maven následujícím příkazem.

mvn clean verify -Prelease-composite -Pcodesigning

Příkaz je spuštěn nástrojem Maven (*mvn*) a za ním následuje spuštění dvou cílů a dvou profilů. Tyto cíle jsou součástí životních cyklů *clean* a *build* a každý z nich pak obsahuje několik fází.

Cíl *clean* má za úkol smazat z projektu adresář *target*, který se vytvoří po každém sestavení.

Cíl *verify* je vázán na stejnojmennou fázi *build* životního cyklu. Provádí verifikaci vytvořených balíčků. Samotné spuštění této fáze, pak předchází spuštění všech předchozích fázi tohoto životního cyklu. Po *verify* pak následuje pouze fáze *install* a *deploy*. *Deploy* fáze je sice v Maven používaná k uvolňování artefaktů ale nástroj CodeCoverage k tomu využívá jiných technologií.

Za definicí obou cílů následují dva profily. Tyto profily se spustí a zůstanou aktivní po celou dobu sestavování. Profil *release-composite* slouží ke spuštění kroků potřebných k nahrání sestaveného nástroje na webový server. Profil *codesigning* pak obsahuje informace, důležité k podpisu obsahu nástroje.

Sestavení pak probíhá v následujícím pořadí: Jednotlivé projekty, nebo pouze složky obsahu-

```
codecoverage.root
codecoverage.bundles
codecoverage.core
codecoverage.ui
codecoverage.features
codecoverage.feature
codecoverage.releng
codecoverage.update
codecoverage.target.neon
```

Obrázek 12: Pořadí sestavení jednotlivých projektů

jící soubor POM, jsou vyčištěny pomocí *maven-clean-plugin*. Ten se provede pokaždé na začátku a až pak následují fáze spuštěné cílem *verify*. Další nástroje a balíčky jsou použity v závislosti na typu projektu. V následující části bude popsáno využití důležitých nástrojů na konkrétních projektech a případných výstup, který se nad daným projektem vygeneruje.

1. **codecoverage.root** - je kořenový projekt obsahující hlavní POM soubor. Nepoužívají se zde žádné nástroje a projekt taky negeneruje žádný výstup. Tento projekt slouží jako začínající bod pro celé sestavení. Je zde tedy spouštěn výše popsaný Maven příkaz.

2. **codecoverage.bundles** - stejně jako kořenový projekt se zde nic nespouští. Jde o složku obsahující propojovací POM soubor a jednotlivé moduly.

3. **codecoverage.core** - první modul již používá některé z balíčků. Nejprve pomocí *tycho-compiler-plugin* a cíle *compile* se provede kompilace zdrojových souborů modulu. Po nastavení cílové platformy pomocí *target-platform-configuration* se zavolá cíl pro zabalení modulu *package-plugin* pomocí *tycho-packaging-plugin*. Ten pak vygeneruje výsledný artefakt **codecoverage.core-1.0.0-SNAPSHOT.jar**(SNAPSHOT verze se nahradí časovým razítkem až při uvolňování). Nakonec proběhne pomocí *maven-jarsigner-plugin* podpis tohoto vygenerovaného artefaktu.

4. **codecoverage.ui** - na druhý modul jsou aplikovány stejné akce jako na modul první. Po kompilaci všech zdrojových kódu a nastavení cílové platformy, se vytvoří výsledný artefakt **codecoverage.ui-1.0.0-SNAPSHOT.jar**. Na závěr pak proběhne podpis obsahu.

5. **codecoverage.features** - následuje složka obsahující *features* a propojovací POM soubor s hlavním POM souborem. Stejně jako u *bundles* i zde se nic nevytváří.

6. **codecoverage.feature** - tento projekt již obsahuje konkrétní *feature*. Kompilace zde není potřeba a jako první se zde nastavuje cílová platforma pomocí *target-platform-configuration*. Následuje zabalení, tentokrát pomocí cíle *package-feature*. Vytvoří se tak **codecoverage.feature-1.0.0-SNAPSHOT.jar**, která se na konci pouze podepíše pomocí *maven-jarsigner-plugin*.

7. **codecoverage.releng** - v dalším kroku znovu následuje složka propojující obsažené projekty z hlavním POM souborem.

8. **codecoverage.update** - v tomto kroku proběhne postupně nahrání celého nástroje na vzdálený repozitář. Nejprve dojde k vytvoření verze doplnění verze pomocí *tycho-packaging-plugin*. Následně se spustí kroky, které nahrají obsah na vzdálený server.

   (a) **1.krok** - pomocí *maven-antrun-plugin* se spustí první krok uvolňování7.3.1. Ten má za úkol stáhnout potřebné kompozitní metadata (pokud nějaké existují).

   (b) **2.krok** - před spuštěním dalšího kroku nahrávání se teprve pomocí *tycho-p2-repository-plugin* vytvoří komponenty, které nebyly doposud vytvořeny(např. **codecoverage.update- 1.0.0.201804200638.zip** s již správně sestavenou verzí). Následně se podepíše zbylý obsah pomocí *maven-jarsigner-plugin*. Nakonec už následuje spuštění druhého kroku 7.3.2 pomocí *tycho-eclipserun-plugin* a cíle *eclipse-run*. Úkolem tohoto kroku je vytvořit či aktualizovat kompozitní metadata stažené v prvním kroku.

   (c) **3.krok** - Poslední krok 7.3.3 je již tvořen samotným uvolněním všech položek na vzdálený server.

9. **codecoverage.target.neon** - posledním část pouze zabalí projekt pro definici cílové platformy pomocí *tycho-packaging-plugin* a cíle *package-target-definition*.

Z těchto jednotlivých kroku je tedy složen celý proces sestavování a uvolňování nástroje na vzdálený server.

Nástroj CodeCoverage je pak pro instalaci do prostředí Eclipse dostupný pod tímto odkazem [14]. Jednotlivé verze tohoto nástroje zabalené do archívu pak pod tímto odkazem [15]. Jednotlivé vydané verze, na které odkazuje kompozitní repozitář [14] jsou pak dostupné pod odkazem [16]. Po vydání nové verze, lze přímo z prostředí Eclipse tento nástroj aktualizovat. Celý nástroj je pak nahrán na Git repozitář [17].

9 Závěr

Cílem této práce, bylo vytvořit podporu do prostředí Eclipse, umožňující měření pokrytí zdrojového kódu, pomocí metriky pokrytí cest. Tento cíl byl splněn a výsledkem je nástroj CodeCoverage. Tento nástroj je rozšířením prototypu nástroje používající danou metriku. Hlavní náplní práce bylo tedy prostudování a následné úpravy původního prototypu. Nejdůležitější a časově nejnáročnější byla celá část sestavování a distribuce CodeCoverage. Největší překážkou v průběhu vývoje bylo množství nových technologií či nástrojů, kterým bylo potřeba porozumět a následně je správně aplikovat.

Jedním z cílů, který nebyl zmíněný v úvodu, bylo navrhnout a implementovat vhodnou prezentací výsledného pokrytí. Konkrétně vytváření grafů pokrytých cest a znázornění nepokrytých cest vůči kódu. Z důvodu zaměření se na cíl zmíněný v úvodu, a tedy vytvoření fungujícího doplňku do prostředí Eclipse, nebyly tyto dva požadavky, mířené na lepší prezentaci výsledků dle použité metriky dosaženy.

Nástroj CodeCoverage je odladěn v prostředí Eclipse ve verzi 4.6. Z důvodu neustálého vývoje platformy Eclipse, lze toto upoutání k jedné verzi považovat za omezení. Z tohoto důvodu nebyl nástroj umístěn v „Eclipse marketplace“, který požaduje odladění na aktuální verzi.

Do budoucna by se mohlo nástroj rozšířit o další definice cílových platforem. Tím by se otevřela možnost instalace tohoto nástroje na různé verze platformy Eclipse. Dále by bylo vhodné umístit nástroj na již zmíněný „Eclipse marketplace“. Tímto krokem by se mnohonásobně zvýšila dostupnost tohoto nástroje. Bylo by taky užitečné vytvořit pro oba moduly, ze kterých se nástroj skládá, testy. Ve vývoji softwaru je tato činnost běžnou praktikou. Prezentace výsledků pokrytí zvýrazněním ve zdrojovém kódu, by mohla být rozšířená o další možnosti, například vykreslením do grafu.

Literatura

[1] MICHÁLEK, Jan. Systém pro měření pokrytí kód testy. Vysoká škola báňská - Technická univerzita Ostrava [online]. 2017 [cit. 2018-03-23]. Dostupné z: `http://hdl.handle.net/10084/119048`

[2] Eclipse Plugins, Bundles and Products. *Eclipse Plugins, Bundles and Products - Eclipse Marketplace* [online]. © Eclipse Foundation, Inc [cit. 2018-04-24]. Dostupné z: `https://marketplace.eclipse.org`

[3] SW Testing concepts. *SW testing concepts* [online]. [cit. 2018-03-24]. Dostupné z: `https://sites.google.com/site/swtestingconcepts/home`

[4] BLEWITT, Dr Alex. Eclipse 4 Plug-in Development by Example Begginner's Guide. Packt Publishing - ebooks Account. ISBN 978-1-78216-032-8.

[5] EclEmma - Java Code Coverage for Eclipse. *EclEmma - Java Code Coverage for Eclipse* [online]. © Mountainminds GmbH & Co. KG and Contributors 2006-2017[cit. 2018-03-24]. Dostupné z: `https://www.eclemma.org/index.html`

[6] Neon Software Repository. *Neon Software Repository* [online]. © Eclipse Foundation, Inc. [cit. 2018-04-17]. Dostupné z: `http://download.eclipse.org/releases/neon/`

[7] Eclipse Tycho for building plug-ins, OSGi bundles and Eclipse applications. *Eclipse, Android and Java training and support* [online]. Lars Vogel, Simon Scholz © vogella GmbH, 2011-2016 [cit. 2018-04-20]. Dostupné z: `http://www.vogella.com/tutorials/EclipseTycho/article.html`

[8] Maven – Welcome to Apache Maven *Maven – Welcome to Apache Maven* [online]. The Apache Software Foundation © 2002–2018 [cit. 2018-04-20]. Dostupné z: `https://maven.apache.org`

[9] Bintray - Download Center Automation & Distribution w. Private Repositories. *Bintray* [online]. JFrog Bintray © 2018 [cit. 2018-04-20]. Dostupné z: `https://bintray.com`

[10] BLEWITT, Dr Alex, 2014. Mastering Eclipse Plug-in Development. Packt Publishing - ebooks Account. ISBN 978-1-78328-779-6.

[11] curl *curl* [online]. Daniel Stenberg © 1996-2018 [cit. 2018-04-20]. Dostupné z: `https://curl.haxx.se`

[12] Bintray REST API. *Bintray* [online]. JFrog Bintray © 2018 [cit. 2018-04-20]. Dostupné z: `https://bintray.com/docs/api/#_content_uploading_publishing`

[13] Publish an Eclipse p2 composite repository on Bintray | Lorenzo Bettini. *Lorenzo Bettini* [online]. [cit. 2018-04-19]. Dostupné z: `http://www.lorenzobettini.it/2016/02/publish-an-eclipse-p2-composite-repository-on-bintray/`

[14] Composite repository for CodeCoverage. *Bintray* [online]. JFrog Bintray © 2018 [cit. 2018-04-20]. Dostupné z: `https://dl.bintray.com/dany17/codecoverage/updates/`

[15] Archive repository for CodeCoverage. *Bintray* [online]. JFrog Bintray © 2018 [cit. 2018-04-20]. Dostupné z: `https://dl.bintray.com/dany17/codecoverage/zipped/`

[16] Releases of CodeCoverage. *Bintray* [online]. JFrog Bintray © 2018 [cit. 2018-04-20]. Dostupné z: `https://dl.bintray.com/dany17/codecoverage/releases/`

[17] Git repository for CodeCoverage. *Git repository for CodeCoverage* [online]. GitLab [cit. 2018-04-24]. Dostupné z: `https://git.cs.vsb.cz/mro0024/codecoverage.git`

10 Přílohy

I. Příloha na CD

Elektronická příloha na disku CD obsahuje složku codecoverage. V této složce se nachází celý nástroj CodeCoverage.